ATARI MAGAZINE

Best of GFA Basic

JANVIER 1993

ATARI magazine

Best of GFA

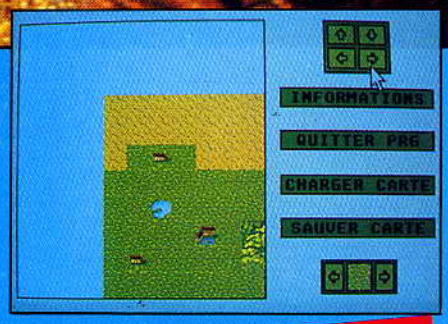
130 pages de programmes commentés

Graphisme programmation de jeux

Effets spéciaux graphiques

Scrolling hard STE

4096 images sur une disquette



Programmez votre propre jeu de rôle

M 1666 - 9301 - 30,00 F

DE L'INITIATION A LA CRÉATION MUSICALE

Vous avez un 1040 STE
Vous êtes un fan de musique...
Equipez-vous à prix réduit

Offre spéciale clavier + logiciel

1990 F TTC seulement
+ 110 F de frais d'expédition.
Livraison sous huitaine

Le clavier BONTEMPI AZ 7500

Dernier-né de la gamme BONTEMPI, il est de loin le plus performant. Il saura vous satisfaire grâce à ses multiples qualités: Clavier MIDI «splitable» • 61 grandes touches • 32 sons «PCM Digital» • 24 rythmes PCM technologie numérique • 7 voies d'accompagnement, polyphonique (12 notes) • 4 effets sonores intégrés. Il s'harmonise idéalement avec le 1040 STE en exploitant toutes ses performances sur le plan musical. Le BONTEMPI AZ 7500 est l'outil qui vous permet de mettre en application vos progrès musicaux.

+

Le logiciel BIG BOSS Piano

Dernier-né de la société RYTHM'N SOFT, BIG BOSS Piano, est le «nec plus ultra» des logiciels musicaux: Cours de piano classique et moderne • clavier guide • défilement des partitions en temps réel • impression des partitions • quantisation • c'est aussi un séquenceur de 25 pistes permettant le chargement simultané de 10 morceaux. Le logiciel BIG BOSS Piano est une méthode éducative simple (assistance permanente) et rapide (10 leçons progressives) pour que le solfège ne soit plus une contrainte insurmontable. Laissez libre cours à votre inspiration et réveillez le musicien qui sommeille en vous.

VOIR BON DE COMMANDE EN PAGE 38

Dans la limite des stocks disponibles.

SOMMAIRE

■ JEU DE ROLE

4 ● **Programmation d'un jeu de rôle (I)**
La programmation d'un jeu de rôle est quelque chose de complexe. Cet article explique certains principes de base utilisés dans des jeux comme *Ultima* et *Dungeon Master*.

11 ● **Programmation d'un jeu de rôle (II)**
La programmation d'un jeu de rôle type *Ultima* passe par la réalisation d'un éditeur de carte permettant de créer le monde dans lequel le joueur doit évoluer.

21 ● **Programmation d'un jeu de rôle (III)**
Voici un petit jeu de rôle où le joueur se déplace sur une carte graphique de grande dimension et fait de dangereuses rencontres.

39 ● **Programmation d'un jeu de rôle (IV)**
Ce programme permet de simuler le déroulement d'un combat entre un personnage et une créature quelconque.

■ JEU D'AVENTURE

51 ● **Jeu d'aventure graphique**
Un petit jeu d'aventure du type *Bat* ou *Final Command*, fonctionnant entièrement à la souris que vous pourrez modifier et compléter pour créer votre propre jeu.

■ PROGRAMMATION DU STE

76 ● **4096 couleurs et scrolling hard STE**
Maîtrisez les nouvelles fonctions graphiques du STE dans vos programmes en *GFA Basic*.

84 ● **Scrolling hard STE horizontal**
Toutes les techniques pour réaliser des scrollings hard STE horizontaux et multidirectionnels.

■ GRAPHISME

70 ● **Bibliothèque de formes de souris**
Une bibliothèque de 40 formes de souris que vous pourrez employer dans vos programmes pour leur donner un air plus sympathique.

92 ● **Effets spéciaux graphiques**
Les effets vidéo permettent d'agrémenter vos applications graphiques et vos jeux. Voici quelques conseils et explications pour bien en comprendre le fonctionnement.

97 ● **4096 images sur une disquette**
Les images des jeux d'arcades sont construites à partir d'éléments graphiques collés les uns contre les autres, rendant ainsi possible le codage d'une image écran en quelques centaines d'octets.

106 ● **Fabriquez vos images par assemblage de sprites**
Nous vous présentons ici une méthode permettant de dessiner des images à partir de sprites, technique beaucoup utilisée par les développeurs de jeux d'aventures pour fabriquer des images avec peu de place mémoire.

114 ● **Instruction RC_COPY**
L'instruction RC_COPY est une instruction graphique du *GFA Basic* 3.0 permettant de réaliser des affichages rapides et des effets spéciaux graphiques originaux.

122 ● **Les opérateurs graphiques logiques**
Les fonctions graphiques logiques du *GFA Basic* permettent de réaliser de nombreux effets graphiques (affichage de sprites, vidéo inverse, effets spéciaux, etc.)

Atari Magazine est édité par ARTIPRESSE
79, avenue Louis Roche - 92330 Gennevilliers - Tél. 40 85 31 90
SIREN 345 365 191 - APE 5120

Gérant, Dr de la publication: Serge Fenez.
Dépôt légal: 1er trimestre 1993
Imprimé par l'Imprimerie Les Mouthieux.

© ARTIPRESSE 1993 - «Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit, des pages publiées dans la présente publication, faite sans l'autorisation de l'éditeur est illicite et constitue une contrefaçon.» (Loi du 11 mars 1957 - art. 40 et 41 et Code Pénal art. 425).

EDITO

BEST OF GFA

Nombre de nouveaux venus à l'informatique ont essayé et adopté le GFA Basic. Atari Magazine a suivi le mouvement en lui consacrant une rubrique particulière que j'ai alimentée régulièrement d'exemples concrets de programmation dédiés au graphisme et à la réalisation des jeux.
Aujourd'hui, Atari Magazine vous propose une compilation des meilleurs articles parus depuis 1990, afin que vous puissiez aller plus loin dans vos réalisations personnelles.

Patrick Leclercq

PROGRAMMATION D'UN JEU DE ROLE (I)

Créez votre propre monde

La programmation d'un jeu de rôle est quelque chose de complexe. Cet article explique certains principes de base utilisés dans des jeux comme Ultima et Dungeon Master.

Description

Dans les jeux de rôle informatiques, un joueur dirige les actions d'une équipe de personnages dans un monde particulier. Elles peuvent être variées: exploration, combat, discussion, commerce, etc.
Les personnages sont définis par des caractéristiques physiques, mentales et des compétences. La manière dont ils agissent sur le monde extérieur dépend de ces dernières (un personnage très fort physiquement se battra bien, un personnage très intelligent apprendra vite). Chaque personnage peut posséder un certain nombre d'objets divers et variés (vêtements et armures pour se protéger des dangers du monde extérieur, armes pour combattre, argent, nourriture, etc.)

Caractéristiques des personnages

Les personnages sont définis par des caractéristiques numériques. Leur nombre peut varier selon le degré de simulation. Je vous propose ici un système simplifié comportant trois caractéristiques physiques (Force, Agilité, Constitution) et trois autres mentales (Intelligence, Volonté, Pouvoir). Le Pouvoir représente la capacité d'un personnage à percevoir la nature véritable de l'univers. C'est une caractéristique qui ne sert que pour la magie (univers médiéval fantastique) ou pour les pouvoirs mentaux (univers de science fiction).

Echelle de référence

Une valeur numérique ne signifie pas grand chose sans une échelle de référence. Pour donner une base de comparaison entre les caractéristiques et la réalité, on considère que la valeur 10 correspond à la moyenne humaine. Un personnage avec une Force, une Dextérité et une Constitution de 10 est un homme tout à fait normal. Des personnages exceptionnels tels que Conan ou Rambo ont une force de 17 ou 18.

Caractéristiques secondaires

Nous venons de voir sont les caractéristiques principales. Un personnage possède aussi un ensemble de caractéristiques secondaires, calculées à partir des principales. Elles définissent les réactions du personnage dans un cas précis. Ces caractéristiques secondaires peuvent être: le nombre de points de vie, le bonus au dommage, la résistance au poison, la résistance aux maladies, etc.
Le nombre de points de vie correspond à la vigueur d'un personnage. Plus il est robuste, plus il peut encaisser de coups avant de mourir. Un personnage chétif (avec une faible Constitution) meurt d'un coup un peu violent. Plus un personnage est fort, plus il cause de dommages à son adversaire. Le bonus au combat simule cet effet. Il s'ajoute aux dommages causés par une arme.
La résistance au poison est un pourcentage qui détermine les chances qu'a un personnage de résister au effets d'un poison. De même, la résistance à la maladie est un pourcentage qui détermine les chances qu'a un personnage de ne pas attraper une maladie.

Stockage des caractéristiques

Les caractéristiques principales et secondaires de tous les personnages sont stockées dans un tableau numérique.

```
DIM def_perso(4,30)
' -----------------
' 1 = Force perso
' 2 = Dextérité perso
' 3 = Constitution
' 4 = Intelligence
' 5 = Volonté
' 6 = Pouvoir
' ----------------
' 7 = Points de vie
' 8 = Bonus dommage
' 9 = résistance au poison
' ...
' -------------
```

Calcul des caractéristiques secondaires

Les caractéristiques secondaires sont calculées par le programme. Le bonus au dommage est en fonction de la Force. Le nombre de points

de vie peut être égal à la Constitution. La résistance au poison peut dépendre de la Constitution et de la Volonté. La valeur du bonus aux dommages est stockée dans un tableau. Cette technique permet de changer facilement la valeur d'un coefficient. Dans cet exemple, les bonus de dommages ne montent pas au delà de 18, mais rien ne nous empêche d'aller plus loin. Il peut servir pour des créatures non humaines (un éléphant a une force de 30).

```
DIM bonus_degat(18)
'
PROCEDURE init_bonus
  LOCAL i
  RESTORE bonus_dommage
  FOR i=1 TO 18
    READ bonus_degat(i)
  NEXT i
RETURN
'
bonus_dommage:
DATA 0,0,0,0,0,0,0,0,0,0,1,1,2,3,4,5,7
Bonus_dommage   = bonus_degat(force)
points_vie      = constitution
resist_poison   = constitution+(volonte/2)
resist_maladies = constitution+volonte
```

Les valeurs sont stockées dans le tableau `def_perso()` à la suite des caractéristiques principales.

Utilisation des caractéristiques secondaires

Lorsqu'un personnage reçoit des dégâts (combat, empoisonnement, chute, etc.), le programme diminue son nombre de points de vie. Si celui-ci tombe à zéro, le personnage est mort. Si le personnage blessé reçoit des soins (guérisseur ou potions), le programme remonte le nombre. Les points de vie pouvant varier en cours de jeu, il faut séparer les points de vie théoriques des points de vie variables. Ceux-ci sont stockés dans un tableau spécial.

```
DIM points_vie(4)
```

Supposons qu'un personnage se fasse empoisonner à un moment quelconque du jeu, le programme doit alors déterminer l'effet du poison: il peut être sans effet sur le personnage ou le foudroyer sur place. La caractéristique Résistance au Poison est un pourcentage. Le programme tire un chiffre compris entre 1 et 100. Si le résultat est inférieur ou égal à la résistance au poison du personnage, tout va bien. Dans le cas contraire, il perd un certain nombre de points de vie (dépendant de la puissance du poison).

Compétences

En plus de ces caractéristiques physiques, un personnage sait faire un certain nombre de choses: se battre, soigner les blessures, évaluer la valeur d'un objet, etc. Ce sont des compétences. Elles sont définies sous la forme de pourcentages. Un personnage débutant a 10% de chance de toucher un ennemi au combat, alors qu'un combattant expérimenté a 90% de chance d'y parvenir. Les compétences les plus courantes dans un jeu de rôle basé sur un univers médiéval-fantastique sont: combattre, soigner, fabriquer des potions, chasser, trouver des herbes, dérober discrètement des objets, déceler les pièges et ouvrir des serrures.

Mémorisation des compétences

Les compétences des personnages sont stockées dans un tableau numérique. Chacune a son numéro d'ordre. Le premier indice du tableau correspond au numéro du personnage.

```
DIM competence(4,100)
' 1 = Compétence combat
' 2 = Compétence soins
' 3 = Compétence trouver herbes
' 4 = Compétence fabriquer potions
' 5 = Compétence voler objets
```

Compétences de base

Le degré de compétence d'un personnage dépend de ses dons naturels. Plus ses caractéristiques sont élevées, plus ses compétences de base sont importantes. La capacité de combat de base peut dépendre de la Force, de la Dextérité et de l'Intelligence. La capacité de base de soins peut dépendre de la Dextérité et de l'Intelligence, etc.
Lors de la création d'un personnage, une routine spécifique détermine ses compétences de base à partir de ses caractéristiques. Dans l'exemple suivant, la compétence de combat de base est calculée en faisant la moyenne de la Force, la Dextérité et l'Intelligence, puis en comparant cette moyenne à une série de chiffres précis. Toutes les compétences de base sont calculées de la même manière.

```
' COMPETENCE DE COMBAT DE BASE
moyenne=(force+dexterite+intelligence)/3
SELECT moyenne
CASE 3 TO 11
  combat_base=0
CASE 12 TO 14
  combat_base=10
CASE 15 TO 16
  combat_base=15
CASE 17
  combat_base=20
CASE 18
  combat_base=30
ENDSELECT
```

Métier des personnages

En plus des compétences de base qui dépendent des caractéristiques, les personnages disposent de compétences professionnelles qu'ils ont appris dans leur jeunesse. Pour simplifier la programmation, on considère qu'il n'existe qu'un petit nombre de métiers. Chacun augmente certaines compétences. Les personnages doivent obligatoirement choisir l'un des métiers proposés. La nature de ces derniers dépend de l'univers du jeu. Par exemple, on peut trouver les métiers suivants dans un univers médiéval-fantastique: guerrier, guérisseur et voleur.

Celui de guerrier apporte des compétences de combat, celui de guérisseur des compétences de soins et le métier de voleur des compétences de combat limitées, ainsi que des facilités pour dérober des objets et trouver des pièges. On trouve souvent le métier de sorcier, mais la gestion de la magie est trop compliquée pour être abordée dans cet article. Nous y reviendrons ultérieurement.
Le tableau `métier()` sert à mémoriser le métier de base des personnages. La procédure `INIT_METIER` initialise automatiquement les compétences professionnelles d'un personnage selon le métier choisi. Les guerriers augmentent leurs compétences de combat de 40%. Les guérisseurs augmentent leurs compétences de soins, pour chercher des herbes et fabriquer des potions de 20%. Les voleurs augmentent leurs compétences de 10% en combat et de 30% sur le vol. Les compétences professionnelles s'ajoutent à celles de base.

```
DIM metier(4)
PROCEDURE init_metier(perso)
  LOCAL metier%
  '
  metier%=metier(perso)
  SELECT metier%
  CASE 1    ! TEST GUERRIER
    ADD competence(perso,1),40
  CASE 2    ! TEST GUERISSEUR
    ADD competence(perso,2),20
    ADD competence(perso,3),20
    ADD competence(perso,4),20
  CASE 3    ! TEST VOLEUR
    ADD competence(perso,1),10
    ADD competence(perso,5),30
  ENDSELECT
RETURN
```

Augmentation des compétences

Les compétences peuvent augmenter avec le temps. Lorsqu'un personnage a vécu quelques aventures (explorer la région, tuer des monstres, trouver des trésors, etc.), il a acquis des points d'expérience utilisables pour améliorer ses capacités. Il doit se rendre dans un lieu précis (université, professeur d'escrime) et échanger ses points d'expérience contre des points de compétences.

Utilisation des compétences

Si un personnage veut frapper un ennemi, le programme doit lire la compétence de combat du personnage et faire un tirage aléatoire. Si le résultat est inférieur ou égal à la compétence de combat, le coup porte. Il faut alors calculer sa force. Le calcul est fait en tenant compte du type d'arme utilisé pour frapper et du type de protection porté par l'ennemi. Lorsqu'un personnage veut accomplir une action particulière, le programme suivra toujours la même démarche: calcul des chances de réussite de l'action, tirage d'une valeur aléatoire pour déterminer si l'action a réussi et traitement du résultat. Ce dernier point débouche sur des situations différentes, selon que l'action est réussie ou non. Prenons un exemple où un personnage essaye de trouver des plantes dans un coin de forêt. En cas de réussite, le programme détermine quelle a été la plante trouvée. En cas d'échec, il ne se passe rien. Et maintenant, prenons un autre exemple où un personnage tente de voler la bourse d'un passant. En cas de réussite, il s'en empare sans problèmes. En cas d'échec, le passant s'aperçoit de la tentative de vol et peut décider de frapper le voleur.

Création d'un personnage

La création d'un personnage se fait en plusieurs étapes: tirage aléatoire des caractéristiques, calcul des caractéristiques secondaires, calcul des compétences de base et définition du métier du personnage.
Les caractéristiques sont tirées au hasard afin de donner des personnages différents. La technique de tirage la plus utilisée met en œuvre 3 tirages différents: on tire 3 fois au hasard un nombre compris entre 1 et 6. On additionne ensuite le tout. Le résultat est un nombre qui varie entre 3 et 18. Cette technique permet d'obtenir un maximum de résultats moyens (9,10,11) et un minimum de chiffres extrêmes (3,4,17,18). Cela correspond bien à la réalité où la majeure partie des gens ont des caractéristiques moyennes et où les extrêmes (force fabuleuse, faiblesse pathologique, très grande dextérité, etc.) sont rares.

Gestion des objets

Elle pose de nombreux problèmes. Il faut définir leurs caractéristiques et les stocker quelque part en mémoire. Les personnages peuvent posséder divers objets et en porter plusieurs sur eux (habillement, armes, armures, etc.)

Classes d'objets

Les objets sont définis par leurs effets sur l'environnement. Une arme est quelque chose qui fait mal. Un habit ou une armure est un objet qui exerce une fonction de protection. La nourriture est un objet qui redonne des forces, etc. Il existe des classes d'objets. Une épée et une hache sont des armes. Un morceau de pain, de la viande grillée, un ragoût de mouton et des épinards sont de la nourriture. Un objet a une classe (Arme, Nourriture, Habits, etc.), et un type (Epée, Hache, Pain, Viande, etc.). Les objets d'une même classe ont la même fonction (toute les armes font des dégâts), mais la manière dont ils agissent dépend de leurs caractéristiques propre (un couteau cause moins de dégâts qu'une épée, un morceau de pain sec nourrit moins bien qu'un morceau de viande, etc.).

Traitement des objets

L'avantage de travailler avec des classes d'objets, c'est que le programme n'a besoin que d'une seule routine de traitement par classe d'objet. Prenons comme exemple la nourriture. La fonction de la nourriture est de redonner force et vitalité à celui qui en mange. Considérons qu'un homme moyen a 20 points de vie. Après un jeûne d'une journée et diverses actions (déplacement, combat, etc.), il a perdu des forces. Il ne lui reste plus que 12 points de vie. Il a le choix entre plusieurs types de nourriture : pain sec, viande, ou fruits. Une portion de pain permet de récupérer 2 points de vie, une de viande 6 points et de fruits 3 points. La nourriture est définie par un nom et un chiffre représentant son effet. Pour la routine de traitement de la nourriture, la seule chose qui compte, c'est l'effet de la nourriture sur la vitalité du personnage. On peut donc s'amuser à

définir un grand nombre de nourritures différentes. On peut rajouter des caractéristiques supplémentaires afin d'améliorer le degré de réalisme: la nourriture peut avoir un prix. Il est possible qu'un personnage préfère manger 3 portions de pain plutôt qu'une de viande si la viande coûte trop cher. Le prix peut être calculé dans n'importe quelle monnaie.

Stockage des caractéristiques des objets en mémoire

Les caractéristiques des objets sont stockées dans des tableaux de données. Chaque classe d'objet a son propre tableau. La classe d'un objet indique le tableau où il faut chercher ces caractéristiques, et le type de l'objet indique la position des caractéristiques dans le tableau.
Les caractéristiques des objets sont définies initialement par le programmeur dans des lignes de DATA. Une routine d'initialisation spécifique charge les informations dans les tableaux de données correspondants. Afin de vous donner quelques exemples, nous allons définir les classes d'objets NOURRITURE, ARME, et ARMURE.

Définition de la nourriture

Les objets de type NOURRITURE sont définis par un nom, un chiffre représentant leur effet sur un personnage et un prix. Il faut deux tableaux pour mémoriser les objets de ce type. Le premier tableau contient le nom, le second la valeur de l'effet "restaurateur" et le prix du produit.

```
DIM def_nourriture$(100)
DIM def_nourriture(100,2)
' def_nourriture$(n)  = Nom nourriture n
' def_nourriture(n,1) = Effet "restaurateur"
' def_nourriture(n,2) = Prix
```

Les objets de type NOURRITURE sont définis dans des lignes de DATA. Les tableaux de données sont remplis par la routine INIT_NOURRITURE. Cette routine stockera le nombre d'objets lus dans la variable globale nb_nourritures.

```
PROCEDURE init_nourriture
LOCAL nom$,n
n=0
RESTORE nourritures
DO
  READ nom$
  EXIT IF nom$="*FIN"
  INC n
  def_nourriture$(n)=nom$
  READ def_nourriture(n,1)
  READ def_nourriture(n,2)
LOOP
nb_nourritures=n
RETURN
nourritures:
DATA PAIN SEC,2,1
DATA VIANDE GRILLEE,6,10
DATA FRUITS,3,5
DATA *FIN
```

Le pain aura le numéro 1, la viande grillée aura le numéro 2, et les fruits porteront le numéro 3. Vous pouvez vous amuser à en définir bien d'autres types.

Définition des armes

Les armes servent à infliger des dégâts. L'importance des dommages dépend du type d'arme (un canif ne fait pas autant de dégâts qu'une hache de bataille). Pour définir le potentiel destructeur des armes, nous nous sommes basés sur le nombre de points de vie moyen des personnages. Avec une moyenne de 10 points de vie par personnage, nous avons défini les armes suivantes : poignard (2 points de dégâts), épée (5 points de dégâts), hache (5 points de dégâts) et hache de bataille (7 points de dégâts). Les armes sont définies par leur nom, leurs points de dégâts et leur prix. Elles sont stockées dans les tableaux def_arme$() et def_arme().

```
DIM def_arme$(100)
DIM def_arme(100,2)
'   def_arme(n)   = Nom arme n
'   def_arme(n,1) = Dégât arme n
'   def_arme(n,2) = Prix arme n
```

Les armes sont définies par le programmeur dans des lignes de DATA. La procédure INIT_ARMES remplie les tableaux def_arme$() et def_arme().

```
PROCEDURE init_armes
LOCAL nom$,n
n=0
RESTORE armes
DO
  READ nom$
  EXIT IF nom$="*FIN"
  INC n
  def_arme$(n)=nom$
  READ def_arme(n,1)
  READ def_arme(n,2)
LOOP
nb_armes=n
RETURN
'
armes:
' DATA nom, degat, prix
DATA POIGNARD,2,30
DATA MASSE,3,20
DATA EPEE,5,200
DATA EPEE LONGUE,6,500
DATA HACHE,5,130
DATA HACHE DE BATAILLE,7,750
DATA *FIN
```

Les créatures rencontrées par les personnages ont des armes. Celles-ci peuvent être des objets manufacturés tels que des épées, des gourdins,

des haches, mais aussi des armes naturelles comme des griffes, des crocs, des tentacules, etc.

```
DATA GRIFFES,4,-1
DATA CROCS,5,-1
DATA TENTACULES,3,-1
DATA *FIN
```

La valeur numérique -1 indique au programme qu'il s'agit d'armes naturelles, que l'on trouve uniquement sur des créatures. Les personnages ne peuvent pas les acheter et s'en servir. Le programme aura besoin d'informations sur les armes en cours de jeu. Les fonctions DOMMAGE_ARME et PRIX_ARME permettent d'obtenir facilement ces informations.

```
FUNCTION dommage_arme(arme)
  RETURN def_arme(arme,1)
ENDFUNC
'
FUNCTION prix_arme(arme)
  RETURN def_arme(arme,2)
ENDFUNC
```

Définition des armures

Les armures sont des systèmes défensifs. Au point de vue simulation, elles encaissent un certain nombre de points de dégâts.
Une armure est définie par un nom, un potentiel de protection et un prix.

```
DIM def_armure$(100)
DIM def_armure(100,2)
'   def_armure$(n)  = Nom armure n
'   def_armure(n,1) = Protection armure n
'   def_armure(n,2) = Prix armure n
```

Lorsqu'un personnage reçoit des points de dégâts suite à une attaque, le programme doit tenir compte de la présence éventuelle d'une armure et diminuer les dégâts en fonction du type d'armure.

```
Dommages reçus = Dommages Reçus - Protection armure
```

Les caractéristiques des armures sont définies dans des lignes de DATA, puis chargées dans les tableaux de données par une routine d'initialisation spécifique.

```
PROCEDURE init_armures
LOCAL nom$,n
'
n=0
RESTORE armures
DO
  READ nom$
  EXIT IF nom$="*FIN"
  INC n
  def_armure$(n)=nom$
  READ def_armure(n,1)
  READ def_armure(n,2)
LOOP
nb_armures=n
RETURN armures:
' -------------
DATA Nom, Protection, Prix
DATA HABITS DE MENDIANT,0,10
DATA VESTE DE TOILE,0,20
DATA VETEMENTS DE CUIR,1,100
DATA ARMURE DE CUIR,2,300
DATA COTTES DE MAILLES,3,1000
DATA ARMURE DE PLAQUES,5,4000
DATA *FIN
```

Les créatures rencontrées par les personnages peuvent avoir des armures. Si ce sont des créatures intelligentes, elles peuvent avoir le même type d'armure que les personnages. Si ce sont des créatures non-intelligentes, l'armure peut faire partie de leur morphologie (écailles, fourrure épaisse, cuir épais, etc.).

```
DATA PEAU EN ECAILLES,2,-1
DATA CUIR EPAIS,2,-1
DATA FOURRURE EPAISSE,1,-1
DATA *FIN
```

La valeur -1 indique que ces armures sont naturelles et les personnages ne peuvent pas les acheter ou les utiliser.
Les fonctions PROTECTION_ARMURE et PRIX_ARMURE fournissent au programme les informations dont il a besoin pour une armure particulière.

```
FUNCTION protection_armure(armure)
RETURN def_armure(armure,1)
ENDFUNC
'
FUNCTION prix_armure(armure)
  RETURN def_armure(armure,2)
ENDFUNC
```

Quelques autres types d'objets

Voici quelques autres types d'objet que vous pouvez définir: bouclier (nom,protection,prix), médicaments (nom, type de maladies soignées, puissance, prix), objets précieux (nom, prix), documents (nom, texte, prix), boissons (nom, effet, prix), anneaux magiques (nom, prix), poisons (nom, puissance du poison, prix), etc.

Possession des objets

Pendant le déroulement du jeu, les personnages possèdent des objets. Ils peuvent les acheter, les voler, les trouver dans un trésor ou sur le cadavre d'un ennemi.
Le programme doit donc conserver la liste des possessions. Celles-ci sont de 2 types: les objets que les personnages portent sur eux (des armes, des vêtements, des armures, etc.) et les objets conservés dans un sac (de la nourriture, des objets précieux, des armes, des parchemins, des médicaments, etc.)

Gestion des sacs

Les sacs peuvent être gérés selon plusieurs techniques. Chacune a ses avantages et ses inconvénients. La méthode la plus simple est d'avoir une variable pour chaque type d'objet. Cela n'est valable qu'avec un petit nombre d'objets différents.

Gestion des armes

Les armes sont mémorisées dans le tableau sac_arme(). Ce tableau est commun à tous les personnages. Le premier indice indique le personnage concerné.
Les dimensions du tableau dépendent du nombre de personnages dans l'équipe et du nombre maximal d'armes (100 dans l'exemple suivant).

```
DIM sac_arme(4,100)
```

La variable sac_arme(n,1) contient le nombre de POIGNARDS possédé par le personnage n. La variable sac_arme(n,2) contient le nombre de MASSES détenu par le personnage n, etc. Le numéro des armes est celui de création dans les lignes de DATA. Le POIGNARD est la première arme définie, il porte le numéro 1. La MASSE est la seconde arme définie, elle porte donc le numéro 2, etc. **Attention**: en changeant la position d'une arme en DATA, on modifie son numéro. Pour augmenter le nombre d'armes possédé par un personnage, il suffit d'incrémenter l'élément de tableau correspondant. La procédure AJOUT_ARME ajoute une arme de type arme aux possessions du personnage perso.

```
PROCEDURE ajout_arme(perso,arme)
  INC sac_arme(perso,arme)
RETURN
```

La fonction TST_ARME permet de connaître le nombre d'armes d'un type quelconque possédé par un personnage.

```
FUNCTION tst_arme(perso,arme)
  RETURN sac_arme(perso,arme)
ENDFUNC
```

La procédure PRENDRE_ARME permet de prendre une arme d'un type précis dans le sac d'un personnage. Pour éviter les erreurs, il est préférable d'utiliser TST_ARME afin de savoir si il y a au moins une arme de ce type dans le sac.

```
PROCEDURE prendre_arme(perso,arme)
  IF sac_arme(perso,arme)<>0
     DEC sac_arme(perso,arme)
  ENDIF
RETURN
```

Gestion des armures

Les armures sont gérées de la même manière que les armes. Le tableau sac_armure() contient les armures des personnages de l'équipe.

```
DIM sac_armure(4,100)
```

La procédure AJOUT_ARMURE permet d'ajouter une armure à un personnage. Elle fonctionne selon le même principe que la procédure AJOUT_ARME. Les routines TST_ARMURE et PRENDRE_ARMURE permettent de tester le nombre d'armures possédées, et de retirer une armure à un personnage.

```
PROCEDURE ajout_armure(perso,armure)
  INC sac_armure(perso,armure)
RETURN
'
FUNCTION tst_armure(perso,armure)
  RETURN sac_armure(perso,armure)
ENDFUNC
'
PROCEDURE prendre_armure(perso,armure)
  IF sac_armure(perso,armure)<>0
     DEC sac_armure(perso,armure)
  ENDIF
RETURN
```

Gestion d'un type d'objet quelconque

Tous les types d'objets peuvent être stockés de la même manière que les armes et les armures: avec un tableau spécifique et quelques routines de manipulation d'objets.

```
DIM sac_bouclier(4,100)
DIM sac_nourriture(4,100)
DIM sac_boisson(4,100)
DIM sac_potion(4,100)
DIM sac_anneaux(4,100)
DIM sac_bijoux(4,100)
```

Cette méthode est simple à mettre en œuvre, mais elle a le défaut de multiplier le nombre de tableaux. La place mémoire perdue peut devenir très importante.

Equipement des personnages

L'équipement des personnages de notre petit programme d'exemple se limite à une arme et à une armure. Cet équipement est stocké dans le tableau equipement_perso(). Lorsqu'une case de ce tableau contient 0, le personnage n'a pas l'équipement correspondant.

```
DIM equipement_perso(4,2)
' equipement_perso(p,1) = Arme perso p
' equipement_perso(p,2) = Armure perso p
```

Le joueur doit pouvoir mettre ou enlever de l'équipement à ces personnages. Il faut donc avoir des routines de manipulation de l'équipement. La procédure METTRE_ARME permet de donner une arme particulière à un personnage. Si celui-ci a déjà une arme en main, elle est stockée dans ses possessions.

```
PROCEDURE mettre_arme(perso,arme)
```

```
  LOCAL old_arme
  '
  old_arme=equipement_perso(perso,1)
  IF old_arm<>0
     @ajout_arme(perso,old_arme)
  ENDIF
  equipement_perso(perso,1)=arme
RETURN
```

La procédure ENLEVER_ARME retire l'arme présente dans les mains d'un personnage et met celle-ci dans ses possessions.

```
PROCEDURE enlever_arme(perso)
  LOCAL arme
  '
  arme=equipement_perso(perso,1)
  IF arme<>0
     @ajout_arme(perso,arme)
     equipement_perso(perso,1)=0
  ENDIF
RETURN
```

La fonction ARME_PERSO fournit au programme l'arme utilisée par un personnage. Si le personnage n'a aucune arme en main, elle donne la valeur 0.

```
FUNCTION arme_perso(perso)
  RETURN equipement_perso(perso,1)
ENDFUNC
```

Les armures peuvent être manipulées de la même manière que les armes. Les procédures METTRE_ARMURE et ENLEVER_ARMURE permettent de revêtir une armure et de la retirer.

```
PROCEDURE mettre_armure(perso,armure)
  LOCAL old_armure
  '
  old_armure=equipement_perso(perso,2)
  IF old_arme<>0
     @ajout_armure(perso,old_armure)
  ENDIF
  equipement_perso(perso,2)=armure
RETURN

PROCEDURE enlever_armure(perso)
  LOCAL armure
  '
  armure=equipement_perso(perso,2)
  IF armure<>0
     @ajout_armure(perso,armure)
     equipement_perso(perso,2)=0
  ENDIF
RETURN
```

La fonction ARMURE_PERSO permet de connaître l'armure portée par le personnage. Si celui-ci n'a pas d'armure, elle donne la valeur 0.

```
FONCTION armure_perso(perso)
  RETURN equipement_perso(perso,2)
ENDFUNC
```

Dans cet exemple, l'équipement est réduit au minimum (une arme et une armure). Rien n'empêche d'améliorer le système et permettre aux personnages d'avoir des poches, un objet à la ceinture, une amulette autour du cou, des anneaux aux doigts, etc. Vous pouvez même détailler l'habillement (pantalons, vestes, chaussures, etc.).

Calcul du dommage définitif

Les combats que livrent les personnages font intervenir les compétences (compétence de combat), les caractéristiques secondaires (bonus de dommage et points de vie) et l'équipement (armes et armures). Un personnage frappant sur une créature quelconque lui cause des dégâts qui sont calculés en fonction du type d'arme, de la force du personnage et de la protection de la créature. La routine DOMMAGE_CREATURE calcule les dégâts reçus par une créature lorsqu'elle reçoit un coup du personnage perso.

```
FUNCTION bonus_perso(perso)
  RETURN def_perso(perso,8)
ENDFUNC
'
FUNCTION dommage_creature(perso,creature)
  LOCAL arme,dommage
  '
  arme=@arme_perso(perso)
  dommage=@dommage_arme(arme)
  ADD dommage,@bonus_perso(perso)
  SUB dommage,@protection_creature(creature)
  IF dommage<0
     dommage=0
  ENDIF
  RETURN dommage
ENDFUNC
```

La routine PROTECTION_CREATURE n'est pas définie dans cet article. Elle fait partie du module de gestion des créatures qui n'est pas détaillé ici, faute de place. Les créatures se gèrent comme les personnages, mais d'une manière moins détaillée.

Pour finir

Cet article vous a présenté quelques notions indispensables pour comprendre le fonctionnement interne des jeux de rôle. C'est un article incomplet car le sujet est vaste.
Nous n'avons pas parlé de la gestion des créatures, des déplacements dans un univers virtuel, des dialogues avec les créatures, des problèmes de représentation graphique, sujets traités dans les autres articles de la série.

POUR TOUTES VOS QUESTIONS, TAPEZ
3615 ATARI

PROGRAMMATION D'UN JEU DE ROLE (II)

Editer vos cartes graphiques

La programmation d'un jeu de rôle façon Ultima passe par la réalisation d'un éditeur capable de définir la carte d'un monde médiéval fantastique.

Notions de base

Dans les jeux de rôle de type *Ultima* ou *Fantaisy* (liste non limitative), le joueur gére les actions d'une équipe de personnages. Celle-ci se déplace sur une grande carte représentant l'ensemble du monde de jeu. Seule une partie de la carte est affichée à l'écran. C'est une sorte de fenêtre de visualisation qui se déplace sur la carte totale. Lorsque les personnages se déplacent, la fenêtre montre la nouvelle position.

Dessin de la carte

L'univers de jeu doit être vaste pour que le joueur ait le plaisir de l'explorer. Cela nous empêche de stocker le dessin de la carte sous la forme d'une image, pour des raisons de place mémoire. Une image de 320x200 pixels en basse résolution occupe 32 Ko. Une image de 1600x800 pixels, équivalent à un espace de 5x4 images écrans, occupe une place mémoire de 640 Ko. C'est beaucoup trop pour un 520 ST et même pour un 1040 ST, en tenant compte de la taille prise par le programme et les autres données graphiques.

Dessin avec éléments graphiques de base

Au lieu de stocker l'image de la carte en mémoire, on peut la redessiner avec des éléments de base. Une forêt peut être dessinée avec 6 ou 7 éléments, une riviére ou un plan d'eau n'ont pas non plus besoin de beaucoup d'éléments différents. A la rigueur, une forêt peut être dessiné avec une seule petite image représentant un arbre, et la montagne avec un seul élément représentant la silhouette d'une cime. C'est le procédé utilisé par bon nombre de logiciels de jeu du commerce. Pour des raisons techniques qui tiennent à l'organisation des mémoires vidéos, les éléments ont le plus souvent une taille de 16x16 pixels ou 32x32 pixels.
Une carte est alors mémorisée sous la forme d'une liste d'éléments graphiques. On peut stocker cette liste dans un tableau numérique à deux dimensions, dont le type dépend du nombre d'éléments possibles. Avec moins de 256 éléments, ce qui est déjà énorme, on peut utiliser un tableau de type BYTE (symbole |), chaque élément du tableau n'occupant qu'un octet en mémoire.

Avec des éléments de 16x16, une carte de 1600x800 pixels peut être stockée dans un tableau de 100x50 éléments, occupant une place mémoire de 5000 octets. Sa déclaration se fait de la manière suivante:

```
OPTION BASE 1
DIM carte|(100,50)
```

L'instruction OPTION BASE 1 force le basic à numéroter ces tableaux à partir de l'indice 1 et non 0. Pour calculer le gain de mémoire entre les deux méthodes, il faut prendre en compte la place mémoire prise par les éléments graphiques.
Un élément graphique de 16x16 pixels stocké dans une variable alphanumérique grâce à l'instruction GET occupe une place de 134 octets, ce calcul ayant été fait avec l'instruction LEN qui détermine la taille d'une variable alphanumérique.

```
GET 1,1,16,16,var$
PRINT LEN(var$)
```

L'expérience montre que l'on peut dessiner des cartes complétes avec bien moins de 200 éléments différents. C'est donc une valeur limite acceptable pour faire quelques calculs. Vous utiliserez certainement moins de 200 éléments pour créer vos propres cartes. Pour la petite histoire, les premiers jeux Ultima ont été dessinés avec seulement quelques dizaines d'éléments. Le stockage de 200 éléments graphiques prend 26800 octets.

```
DIM element$(200)
```

En additionnant la place prise par le codage de la carte et par 200 éléments graphique, on arrive à un total de 31800 octets (5000+26800). Dans notre cas particulier, le gain de place apporté par le codage de la carte est de 20, mais s'améliore considérablement si la carte est plus grande que 100x50 éléments ou si le nombre d'objets graphique est plus petit que 200.
Cette technique offre un gros avantage pour le développement d'un logiciel: la carte est facile à dessiner une fois que les éléments de base sont créés.

Même un non-graphiste peut réaliser une jolie carte à partir de beaux éléments de base.

Dessin de la carte Ultima

La portion de carte affichée par les jeux *Ultima* a une dimension de 176x176 pixels, et est constitué de 11x11 élements de 16x16 pixels. La position de l'équipe de personnages est visualisée par une silhouette en noir affichée au milieu de l'écran.
Lors des déplacements, l'image de la carte est redessinée de manière à ce que l'équipe soit toujours au centre de l'image visible.

Programmation d'un système de déplacement style Ultima

Pour réaliser un systéme de déplacement similaire à celui d'*Ultima* en *GFA Basic*, il faut résoudre un certain nombre de problémes techniques. Nous avons déjà vu que le codage de la carte peut être stocké dans un tableau numérique et les éléments graphiques de base dans un tableau alphanumérique, mais il reste bien des questions à étudier. Comment est-ce que les éléments graphiques sont stockés dans le tableau alphanumérique? Comment afficher rapidement la carte aussi bien sur un STF que sur un STE? Comment stocker les différentes données sur disque? Comment gérer les déplacements de l'équipe sur la carte?
Toutes ces questions sont trop difficiles à gérer dans un seul article, c'est pourquoi ce texte est séparé en deux parties: la première est consacrée à la création d'une carte et la seconde traite de son utilisation pour gérer les déplacements de l'équipe.

Définition des éléments graphiques

On suppose que les éléments graphiques de base ont été dessinés par un graphiste avec un logiciel de dessin type *Degas Elite* ou équivalent. Nos précieux éléments sont donc stockés sur disque dans une image au format *Degas*. Le programme GENELEM.GFA lit l'image *Degas* ELEMENTS.PI1 et stocke les éléments graphiques dans un fichier binaire ELEMENTS.BIN réutilisable par un autre programme.
La structure du fichier binaire est la suivante: 32 octets contenant la palette de couleur des éléments graphiques, un octet unique contenant le nombre d'élements du fichier et le codage des éléments, chacun occupant 134 octets. La position (x,y) des éléments sur l'image est définie dans une série de DATA à la fin du programme, la fin des données étant signalé par la valeur -1.

```
' ********************************
' *  GENERATEUR FICHIER BINAIRE  *
' *     ELEMENTS GRAPHIQUES      *
' ********************************
RESERVE 100000
DIM element$(200)
nb_elements%=0
palette$=SPACE$(32)
'
@load_degas("ELEMENTS.PI1")
@saisie_elements
@sauve_elements("ELEMENTS.BIN")
END
'
' **********************
' *  CHARGEMENT IMAGE  *
' *     DEGAS ELITE    *
' **********************
' ***********************************
' * La palette est stockée dans la *
' * variable globale palette$      *
' ***********************************
PROCEDURE load_degas(nom$)
  palette$=SPACE$(32)
  OPEN "I",#1,nom$
  SEEK #1,2
  BGET #1,VARPTR(palette$),32
  VOID XBIOS(6,L:VARPTR(palette$))
  BGET #1,XBIOS(2),32000
  CLOSE #1
RETURN
'
' ********************************
' *  GENERATION FICHIER BINAIRE  *
' *  ELEMENTS GRAPHIQUES         *
' ********************************
PROCEDURE sauve_elements(fichier$)
  LOCAL i%
  OPEN "o",#1,fichier$
  BPUT #1,VARPTR(palette$),32
  OUT #1,nb_elements%
  FOR i%=1 TO nb_elements%
    BPUT #1,VARPTR(element$(i%)),134
  NEXT i%
  CLOSE #1
RETURN
'
'
' *************************
' *     CHARGEMENT DES    *
' *  ELEMENTS GRAPHIQUES  *
' *************************
PROCEDURE saisie_elements
  LOCAL n%
  LOCAL px%,py%
  LOCAL px2%,py2%
  '
  n%=0
  RESTORE def_elements
  DO
    READ px%
    EXIT IF px%=-1
    READ py%
    INC n%
    px2%=px%+15
    py2%=py%+15
    GET px%,py%,px2%,py2%,element$(n%)
  LOOP
  nb_elements%=n%
```

```
RETURN
'
' ************************
' *     POSITION DES     *
' *  ELEMENTS GRAPHIQUES *
' ************************
def_elements:
DATA 1,1
DATA 18,1
DATA 35,1
DATA 52,1
DATA 69,1
DATA 86,1
DATA 1,18
DATA 18,18
DATA 35,18
DATA 52,18
DATA 69,18
'
DATA -1
```

Programme EDITCART

Le programme EDITCART est un éditeur de carte virtuelle écrit entiérement en *GFA Basic*. Il sauve les cartes éditées dans des fichiers binaires, pouvant être relus et exploités par d'autres programmes.
Ses fontions sont: lecture d'une carte à partir du disque, sauvegarde de la carte courante sur disque, déplacement sur la carte, modification d'un élément quelconque de la carte, sélection d'un élément graphique, affichage d'une boîte d'informations et option pour quitter le programme. Le tout est entiérement géré à la souris.
Attention: l'éditeur à besoin d'un fichier d'éléments graphiques pour fonctionner. Exécutez le programme GENELEM avant d'utiliser EDITCART. Le nom du fichier d'élements graphiques est ELEMENTS.BIN par défaut, mais vous pourrez le modifier dans le code source d'EDITCART, ou ajouter une fonction permettant de charger n'importe quel fichier d'éléments graphiques.

La simplicité n'est pas de ce monde

Initialement, cet éditeur a été conçu pour être simple à utiliser et à comprendre. Hélas, il s'est avéré impossible d'écrire des routines d'affichages simples et performantes aussi bien sur STE que STF (dommage que tous les ST ne soient pas dotés de blitter). Les routines d'affichage de EDITCART utilisent donc des techniques de programmation un peu complexes à comprendre pour un programmeur débutant.

Un système d'affichage simple

La première méthode qui vient à l'esprit est de réafficher entiérement l'écran après chaque déplacement. Dans l'exemple qui suit, le codage de la carte est stocké dans le tableau CARTE|() et les éléments graphiques de base dans le tableau ELEMENT$(). La procédure @aff_carte dessine la carte sur l'écran.
La position du morceau de carte visible par rapport à la carte virtuelle est contenu dans les variables ligne_carte et colonne_carte. En mettant ligne_carte et colonne_carte à 1, le côté haut gauche de la carte est affiché sur l'écran.
Pour déplacer la carte d'une position vers la droite, il faut incrémenter la variable colonne_ligne de 1 et appeler de nouveau @aff_carte.
Pour déplacer la carte d'une position vers le bas, il faut incrémenter ligne_carte et exécuter @aff_carte.
Remarque: la position (1,1) correspond au premier indice du tableau contenant la carte, grâce à l'instruction OPTION BASE 1. Sans celle-ci, le premier indice du tableau a la position (0,0).
Le morceau de carte visible est affiché à l'écran à la position définie par les variables xcarte% et ycarte%. Etant donné la structure interne de la mémoire vidéo et la manière dont fonctionnent les instructions graphiques du *GFA Basic*, il est préférable d'utiliser une position d'affichage en x multiple de 16, afin d'accélérer la vitesse d'exécution.

```
OPTION BASE 1
'
xcarte%=16
ycarte%=5
'
ligne_carte%=30
colonne_carte%=67
'
@aff_carte
END

PROCEDURE aff_carte
  LOCAL l%,c%
  LOCAL ligne%
  LOCAL colonne%
  LOCAL num%
  LOCAL posx%
  LOCAL posy%
  '
  posy%=ycarte%
  ligne%=ligne_carte%
  FOR l%=1 TO 11
    posx%=xcarte%
    colonne%=colonne_carte%
    FOR c%=1 TO 11
      num%=carte|(colonne%,ligne%)
      PUT posx%,posy%,element$(num%)
      INC colonne%
      ADD posx%,16
    NEXT c%
    INC ligne%
    ADD posy%,16
  NEXT l%
RETURN
```

Pour gérer les déplacements, il suffit d'avoir une série de procédures modifiant les variables de position en fonction de la direction de déplacement. Pour des raisons de sécurité, ces procédures doivent contrôler que le déplacement ne se fasse pas en dehors des limites de la carte.

```
PROCEDURE aller_haut
  IF ligne_carte%<>1
    DEC ligne_carte%
    @aff_carte
  ENDIF
RETURN

PROCEDURE aller_bas
  IF ligne_carte%<>ligne_max%
    INC ligne_carte%
    @aff_carte
  ENDIF
RETURN

PROCEDURE aller_gauche
  IF colonne_carte%<>1
    DEC colonne_carte%
    @aff_carte
  ENDIF
RETURN

PROCEDURE aller_droite
  IF colonne_carte%<>
    INC colonne_carte%
    @aff_carte
  ENDIF
RETURN
```

Ce système est parfait sur le papier, mais une fois mis en oeuvre, il se révéle trop lent pour une utilisation pratique, cas de figure fréquent en informatique. La solution passe par une amélioration du principe de base.

Un redessin partiel de la carte

On peut gagner du temps en ne redessinant qu'une partie de la carte. Lors d'un déplacement vers la droite, la colonne 1 disparaît de l'écran et les colonnes 2 à 11 sont décalées d'une position vers la gauche, le programme ne dessine vraiment que la nouvelle colonne 11. De la même manière, lors d'un déplacement vers la droite, seule la colonne 1 est nouvelle. Puisqu'une partie de la carte est juste décalée, au lieu de le redessiner, on peut la déplacer en une seule opération graphique. L'une des solutions possible est de lire la portion de carte avec l'instruction GET et de l'afficher à sa nouvelle position avec PUT.

Cette technique fonctionne, mais présente un inconvénient d'ordre esthétique. L'oeil humain est un outil très perfectionné qui est capable d'analyser et de décomposer la structure d'un mouvement rapide. C'est souvent un avantage, mais dans notre cas particulier, c'est un inconvénient: si la routine d'affichage travaille en deux étapes, cela produit un effet graphique nuisible visible. Pour éviter cela, il faut que l'affichage s'effectue en une seule fois et le plus rapidement possible.

La seule manière d'obtenir un affichage rapide est de réaliser les opérations graphiques en mémoire et de les recopier rapidement sur l'écran en une seule opération. Afin d'obtenir un résultat graphique encore meilleur, le programme EDITCART utilise une variante de cette technique: il dessine la carte dans un écran virtuel, puis change l'adresse de la mémoire écran par l'adresse de cet écran virtuel. Ainsi, on évite de recopier l'image sur l'écran.

La fonction XBIOS(5) permet de changer l'adresse de la mémoire écran. Sur un STE cette adresse peut être quelconque, alors que sur un STF, elle doit obligatoirement être un multiple de 256. L'adresse initiale de l'écran est multiple de 256. Il faut créer une zone de 32000 octets pour l'écran virtuel. L'instruction MALLOC permet d'allouer une zone mémoire de n octets. Cette zone mémoire commence obligatoirement à une adresse paire. Pour obtenir une adresse multiple de 256, il faut déclarer avec MALLOC une zone de 32255 octets et masquer les deux derniers bits de l'adresse. Le résultat est une adresse multiple de 256 qui porte sur une zone mémoire de 32000 octets contenus dans la zone mémoire de 32255 octets déclarée par le programme. C'est un peu compliqué, mais cela marche bien.

```
adr_tampon%=MALLOC(32255)
adr2%=(adr_tampon%+255) AND &HFFFFFF00
```

Les variables adr1% et adr2% contiennent l'adresse des deux écrans. La variable adr1% contient l'adresse initiale de la mémoire écran et adr2% contient l'adresse de l'écran virtuel de 32000 octets. L'affichage se fait en bascule. Lorsque l'adresse de la mémoire écran est positionnée sur adr1%, l'image de la nouvelle carte est dessinée dans adr2%. Lorsque l'adresse de la mémoire écran est positionnée sur adr1%, l'image de la nouvelle carte est dessinée dans adr1%.

Lorsque le programme a terminé de dessiner une image en mémoire, il change l'adresse de la mémoire écran grâce à l'instruction XBIOS(5). Attention: la fonction XBIOS(5) peut être appelée à tous moment par le programme, mais le gestionnaire d'écran ne change l'adresse de la mémoire écran que lorsque l'image courante a ét éentiérement dessinée. Si vous utilisez trop rapidement l'instruction XBIOS(5), seul le dernier changement de mémoire d'écran sera pris en compte. Pour éviter ce probléme, aprés avoir utilisé XBIOS(5), il faut attendre que le gestionnaire d'écran ait affiché entièrement l'image écran. Ce test est réalisé avec l'instruction VSYNC. Celle-ci bloque l'exécution du programme tant que le gestionnaire d'écran affiche l'image courante (il y a 50 images affichés par seconde).

```
VOID XBIOS(5)
VSYNC
```

Format de sauvegarde des cartes

Les cartes générées par EDITCART ont une taille de 30 lignes de 60 colonnes. Pour changer la taille, il faut modifier les variables nb_colonnes et nb_lignes sans dépasser les limites du tableau carte(200,75).

Les cartes d'EDITCART ont le format suivant: 19 octets contenant la chaîne "CARTE JdR ATARI MAG", un octet contenant le nombre de colonnes, un octet contenant le nombre de lignes et le codage de la carte (la taille de ce codage est égal à nb_colonnesxnb_lignes octets). La chaîne est un en-tête d'identification. Si vous tentez de lire un fichier n'ayant pas d'en-tête, le programme affiche un message d'erreur. Le nom d'une carte doit impérativement avoir une extension .CAR, ne l'oubliez pas au moment de taper de la carte à sauver (c'est l'unique moment où l'éditeur utilise le clavier).

```
' ******************************
' *       EDITEUR DE CARTE     *
' *       POUR JEU DE ROLE     *
' *         STYLE ULTIMA       *
' ******************************
' *  (C) 1991 ATARI MAGAZINE   *
' *  (C) 1991 PATRICK LECLERCQ *
' ******************************
'
RESERVE 100000
'
OPTION BASE 1
DIM element$(200)
'
adr_tampon%=MALLOC(32255)
adr2%=(adr_tampon%+255) AND &HFFFFFF00
adr1%=XBIOS(2)
'
nb_colonnes%=60
nb_lignes%=30
DIM carte|(200,75)
'
@main
VOID XBIOS(5,L:adr1%,L:adr1%,-1)
CLS
VOID MFREE(adr_tampon%)
END
'
' ------------------------------
PROCEDURE set_ecr_log(adr_ecran%)
  VOID XBIOS(5,L:adr_ecran%,L:-1,-1)
RETURN
'
PROCEDURE set_ecr_phys(adr_ecran%)
  VOID XBIOS(5,L:-1,L:adr_ecran%,-1)
RETURN
'
' ------------------------------
' ******************************
' *  ATTENTE ARRET CLIC SOURIS  *
' ******************************
PROCEDURE att0clic
  DO
    EXIT IF MOUSEK=0
  LOOP
RETURN
'
PROCEDURE souris_fleche
  DEFMOUSE 0
RETURN
'
PROCEDURE souris_abeille
  DEFMOUSE 2
RETURN
'
' ******************************
' *  TEST CLIC SUR ZONE ECRAN  *
' ******************************
FUNCTION tstzone(x%,y%,px%,py%,tx%,ty%)
  LOCAL result%
  '
  result%=1
  '
  IF x%<px%
    result%=0
  ENDIF
  IF x%>px%+tx%-1
    result%=0
  ENDIF
  IF y%<py%
    result%=0
  ENDIF
  IF y%>py%+ty%-1
    result%=0
  ENDIF
  '
  RETURN result%
ENDFUNC
'
' **********************
' *  EFFACEMENT ECRAN  *
' **********************
PROCEDURE cls_ecran(c%)
  DEFFILL c%
  PBOX 0,0,319,199
RETURN
'
' ******************************
' *  AFFICHAGE CADRE GRAPHIQUE  *
' ******************************
PROCEDURE cadre(px%,py%,tx%,ty%,f%,b%)
  LOCAL px2%,py2%
  '
  px2%=px%+tx%-1
  py2%=py%+ty%-1
  DEFFILL f%
  PBOX px%,py%,px2%,py2%
  COLOR b%
  BOX px%,py%,px2%,py2%
RETURN
'
' ************************
' *  AFFICHAGE MESSAGE   *
' *  DANS BOITE DE CLIC  *
' ************************
PROCEDURE aff_boite(px%,py%,tx%,ty%,f%,b%,m$)
  LOCAL px2%,py2%
  LOCAL xaff%,yaff%
  '
  px2%=px%+tx%-1
  py2%=py%+ty%-1
  '
  xaff%=px%+(tx%-LEN(m$)*8)/2
  yaff%=py%+6+(ty%-7)/2
```

```
  '
  DEFFILL f%
  PBOX px%,py%,px2%,py2%
  COLOR b%
  BOX px%,py%,px2%,py2%
  '
  TEXT xaff%,yaff%,m$
RETURN
'
' ****************************************
' *  CHARGEMENT DU FICHIER BINAIRE       *
' *  CONTENANT LES ELEMENTS GRAPHIQUES   *
' ****************************************
PROCEDURE charge_elements(fichier$)
  LOCAL n%
  LOCAL palette$
  '
  palette$=SPACE$(32)
  OPEN "I",#1,fichier$
  BGET #1,VARPTR(palette$),32
  VOID XBIOS(6;L:VARPTR(palette$))
  nb_elements%=INP(#1)
  FOR n%=1 TO nb_elements%
    element$(n%)=SPACE$(134)
    BGET #1,VARPTR(element$(n%)),134
  NEXT n%
  CLOSE #1
RETURN
'
'
' ********************************
' *  INITIALISATION DE LA CARTE  *
' ********************************
PROCEDURE init_carte
  ARRAYFILL carte|(),1
RETURN
'
'
' ****************************
' *  AFFICHAGE IMAGE EDITEUR *
' ****************************
PROCEDURE aff_ecran
  @cls_ecran(1)
  '
  @cadre(xcarte%-1,ycarte%-1,178,178,1,0)
  DEFTEXT 0
  '
  @cadre(226,11,48,36,2,0)
  @aff_boite(229,14,20,14,3,0,CHR$(1))
  @aff_boite(229,30,20,14,3,0,CHR$(4))
  @aff_boite(251,14,20,14,3,0,CHR$(2))
  @aff_boite(251,30,20,14,3,0,CHR$(3))
  '
  @aff_boite(195,56,110,13,2,0,"INFORMATIONS")
  @aff_boite(195,83,110,13,2,0,"QUITTER PRG")
  @aff_boite(195,111,110,13,2,0,"CHARGER CARTE")
  @aff_boite(195,138,110,13,2,0,"SAUVER CARTE")
  '
  @cadre(223,164,54,22,2,0)
  @aff_boite(225,166,14,18,3,0,CHR$(4))
  @cadre(241,166,18,18,2,0)
  @aff_boite(261,166,14,18,3,0,CHR$(3))
  @aff_element_courant
RETURN
'
'
' *********************
' *  AFFICHAGE DE LA  *
' *   CARTE VISIBLE   *
' *********************
PROCEDURE aff_carte
  LOCAL l%,c%
  LOCAL ligne%
  LOCAL colonne%
  LOCAL num%
  LOCAL posx%
  LOCAL posy%
  '
  posy%=ycarte%
  ligne%=ligne_carte%
  FOR l%=1 TO 11
    posx%=xcarte%
    colonne%=colonne_carte%
    FOR c%=1 TO 11
      num%=carte|(colonne%,ligne%)
      PUT posx%,posy%,element$(num%)
      INC colonne%
      ADD posx%,16
    NEXT c%
    INC ligne%
    ADD posy%,16
  NEXT l%
RETURN
'
' *********************
' *   AFFICHAGE DE    *
' *  ELEMENT COURANT  *
' *********************
PROCEDURE aff_element_courant
  LOCAL old_ecr%
  '
  old_ecr%=XBIOS(3)
  @set_ecr_log(adr1%)
  PUT 242,167,element$(element_courant%)
  @set_ecr_log(adr2%)
  PUT 242,167,element$(element_courant%)
  set_ecr_log(old_ecr%)
RETURN
'
'
' ************************************
' *  GESTION PERMUTATION DES ECRANS  *
' ************************************
PROCEDURE preparation_ecr_travail
```

```
  IF etat%=1
    ecr_visible%=adr1%
    ecr_travail%=adr2%
  ELSE
    ecr_visible%=adr2%
    ecr_travail%=adr1%
  ENDIF
  etat%=-etat%
  @set_ecr_log(ecr_travail%)
RETURN
'
' ******************************
' *  VISUALISATION DE LA CARTE  *
' ******************************
PROCEDURE visualisation_ecr_travail
  @set_ecr_phys(ecr_travail%)
  VSYNC
RETURN
'
'
' ******************************
' *  DEPLACEMENT VERS LE HAUT  *
' ******************************
PROCEDURE aller_haut
  LOCAL ecr1%,x1%,y1%
  LOCAL ecr2%,x2%,y2%
  LOCAL px%,py%
  LOCAL c%,colonne%
  LOCAL n%
  '
  IF ligne_carte%<>1
    DEC ligne_carte%
    HIDEM
    @preparation_ecr_travail
    ecr1%=ecr_visible%
    x1%=xcarte%
    y1%=ycarte%
    ecr2%=ecr_travail%
    x2%=xcarte%
    y2%=ycarte%+16
    RC_COPY ecr1%,x1%,y1%,176,160 TO ecr2%,x2%,y2%
    colonne%=colonne_carte%
    px%=xcarte%
    py%=ycarte%
    FOR c%=1 TO 11
      n%=carte|(colonne%,ligne_carte%)
      PUT px%,py%,element$(n%)
      INC colonne%
      ADD px%,16
    NEXT c%
    @visualisation_ecr_travail
    SHOWM
  ENDIF
RETURN
'
' ******************************
' *  DEPLACEMENT VERS LE BAS  *
' ******************************
PROCEDURE aller_bas
  LOCAL ecr1%,x1%,y1%
  LOCAL ecr2%,x2%,y2%
  LOCAL px%,py%
  LOCAL c%,colonne%
  LOCAL n%
  '
  IF ligne_carte%<>nb_lignes%-10
    INC ligne_carte%
    HIDEM
    @preparation_ecr_travail
    ecr1%=ecr_visible%
    x1%=xcarte%
    y1%=ycarte%+16
    ecr2%=ecr_travail%
    x2%=xcarte%
    y2%=ycarte%
    RC_COPY ecr1%,x1%,y1%,176,160 TO ecr2%,x2%,y2%
    colonne%=colonne_carte%
    px%=xcarte%
    py%=ycarte%+160
    FOR c%=1 TO 11
      n%=carte|(colonne%,ligne_carte%+10)
      PUT px%,py%,element$(n%)
      INC colonne%
      ADD px%,16
    NEXT c%
    @visualisation_ecr_travail
    SHOWM
  ENDIF
RETURN
'
' ********************************
' *  DEPLACEMENT VERS LA GAUCHE  *
' ********************************
PROCEDURE aller_gauche
  LOCAL ecr1%,x1%,y1%
  LOCAL ecr2%,x2%,y2%
  LOCAL px%,py%
  LOCAL l%,ligne%
  LOCAL n%
  '
  IF colonne_carte%<>1
    DEC colonne_carte%
    HIDEM
    @preparation_ecr_travail
    ecr1%=ecr_visible%
    x1%=xcarte%
    y1%=ycarte%
    ecr2%=ecr_travail%
    x2%=xcarte%+16
    y2%=ycarte%
    RC_COPY ecr1%,x1%,y1%,160,176 TO ecr2%,x2%,y2%
    ligne%=ligne_carte%
    px%=xcarte%
    py%=ycarte%
```

```
    FOR l%=1 TO 11
      n%=carte|(colonne_carte%,ligne%)
      PUT px%,py%,element$(n%)
      INC ligne%
      ADD py%,16
    NEXT l%
    @visualisation_ecr_travail
    SHOWM
  ENDIF
RETURN
'
' ********************************
' *  DEPLACEMENT VERS LA DROITE  *
' ********************************
PROCEDURE aller_droite
  LOCAL ecr1%,x1%,y1%
  LOCAL ecr2%,x2%,y2%
  LOCAL px%,py%
  LOCAL l%,ligne%
  LOCAL n%
  '
  IF colonne_carte%<>nb_colonnes%-10
    INC colonne_carte%
    HIDEM
    @preparation_ecr_travail
    ecr1%=ecr_visible%
    x1%=xcarte%+16
    y1%=ycarte%
    ecr2%=ecr_travail%
    x2%=xcarte%
    y2%=ycarte%
    RC_COPY ecr1%,x1%,y1%,160,176 TO ecr2%,x2%,y2%
    ligne%=ligne_carte%
    px%=xcarte%+160
    py%=ycarte%
    FOR l%=1 TO 11
      n%=carte|(colonne_carte%+10,ligne%)
      PUT px%,py%,element$(n%)
      INC ligne%
      ADD py%,16
    NEXT l%
    @visualisation_ecr_travail
    SHOWM
  ENDIF
RETURN
'
' ********************************
' *  SAUVEGARDE CARTE SUR DISQUE  *
' *  FONCTION SYSTEME             *
' ********************************
PROCEDURE exec_sauver_carte(nom$)
  LOCAL entete$
  LOCAL l%,c%
  '
  @souris_abeille
  OPEN "O",#1,nom$
  entete$="CARTE JdR ATARI MAG"
  BPUT #1,VARPTR(entete$),19
  OUT #1,nb_colonnes%
  OUT #1,nb_lignes%
  FOR c%=1 TO nb_colonnes%
    FOR l%=1 TO nb_lignes%
      OUT #1,carte|(c%,l%)
    NEXT l%
  NEXT c%
  CLOSE #1
  @souris_fleche
RETURN
'
' ********************************
' *  SAUVEGARDE CARTE SUR DISQUE  *
' *  INTERFACE UTILISATEUR        *
' ********************************
PROCEDURE sauver_carte
  LOCAL nom$,path$
  LOCAL b%
  LOCAL fond$
  '
  HIDEM
  SGET fond$
  SHOWM
  path$=DIR$(0)+"\*.CAR"
  FILESELECT path$," ",nom$
  IF nom$<>""
    IF EXIST(nom$)
      m$="NOM DE FICHIER EXISTANT. "
      m$=m$+"|CONTINUER SAUVEGARDE ? |"
      ALERT 0,m$,1,"OUI|NON",b%
      IF b%=1
        @exec_sauver_carte(nom$)
      ENDIF
    ELSE
      @exec_sauver_carte(nom$)
    ENDIF
  ENDIF
  HIDEM
  SPUT fond$
  SHOWM
RETURN
'
'
' ***************************************
' *  CHARGER CARTE A PARTIR DU DISQUE  *
' ***************************************
PROCEDURE charger_carte
  LOCAL nom$,path$
  LOCAL entete$
  LOCAL l%,c%
  LOCAL b%
  LOCAL m$
  LOCAL fond$
  '
  SGET fond$
  path$=DIR$(0)+"\*.CAR"
```

```
  FILESELECT path$," ",nom$
  SPUT fond$
  IF nom$<>""
    @souris_abeille
    OPEN "I",#1,nom$
    entete$=SPACE$(19)
    BGET #1,VARPTR(entete$),19
    IF entete$<>"CARTE JdR ATARI MAG"
      CLOSE #1
      @souris_fleche
      m$="CE FICHIER NE CONTIENT|PAS DE CARTE"
      ALERT 0,m$,1," OK ",b%
    ELSE
      nb_colonnes%=INP(#1)
      nb_lignes%=INP(#1)
      FOR c%=1 TO nb_colonnes%
        FOR l%=1 TO nb_lignes%
          carte|(c%,l%)=INP(#1)
        NEXT l%
      NEXT c%
      CLOSE #1
      ligne_carte%=1
      colonne_carte%=1
      @aff_carte
      @souris_fleche
    ENDIF
  ENDIF
RETURN
'
' ************************************
' *  INCREMENTATION ELEMENT COURANT  *
' ************************************
PROCEDURE inc_element
  IF element_courant%<>nb_elements%
    INC element_courant%
    @aff_element_courant
    @attOclic
  ENDIF
RETURN
'
' ************************************
' *  DECREMENTATION ELEMENT COURANT  *
' ************************************
PROCEDURE dec_element
  IF element_courant%<>1
    DEC element_courant%
    @aff_element_courant
    @attOclic
  ENDIF
RETURN
'
' ************************************
' *  CLIC SUR UN ELEMENT DE LA CARTE  *
' ************************************
PROCEDURE clic_carte(xm%,ym%)
  LOCAL x%,y%
  LOCAL ligne_clic%
  LOCAL colonne_clic%
  LOCAL ligne%
  LOCAL colonne%
  LOCAL old_element%
  '
  x%=xm%-xcarte%
  y%=ym%-ycarte%
  ligne_clic%=(y%/16)
  colonne_clic%=(x%/16)
  ligne%=ligne_clic%+ligne_carte%
  colonne%=colonne_clic%+colonne_carte%
  '
  old_element%=carte|(colonne%,ligne%)
  IF old_element%<>element_courant%
    carte|(colonne%,ligne%)=element_courant%
    posx%=colonne_clic%*16+xcarte%
    posy%=ligne_clic%*16+ycarte%
    HIDEM
    PUT posx%,posy%,element$(element_courant%)
    SHOWM
  ENDIF
RETURN
'
' ******************************
' *  OPTION 'QUITTER LE PRG'  *
' ******************************
PROCEDURE quitter_programme
  LOCAL m$,t$,b%
  '
  sortie%=0
  m$="VOULEZ-VOUS VRAIMENT  |QUITTER CE PRG ?"
  choix$="OUI|NON"
  ALERT 0,m$,2,choix$,b%
  IF b%=1
    sortie%=1
  ENDIF
  SHOWM
RETURN
'
' *************************
' *  OPTION INFORMATIONS  *
' *************************
PROCEDURE aff_infos
  LOCAL m$,b%
  '
  m$="EDITEUR DE CARTE EN GFA BASIC "
  m$=m$+"|(C) 1991 ATARI MAGAZINE"
  m$=m$+"|(C) 1991 PATRICK LECLERCQ"
  ALERT 0,m$,1," OK ",b%
RETURN
'
' *********************
' *  GESTION EDITEUR  *
' *********************
PROCEDURE gestion_editeur
  LOCAL xm%,ym%,km%
  '
```

```
  DO
    DO
      MOUSE xm%,ym%,km%
      EXIT IF km%<>0
    LOOP
    '
    IF @tstzone(xm%,ym%,195,83,110,13)
      @quitter_programme
    ENDIF
    EXIT IF sortie%=1
    '
    IF @tstzone(xm%,ym%,195,56,110,13)
      @aff_infos
    ENDIF
    '
    IF @tstzone(xm%,ym%,195,111,110,13)
      @charger_carte
    ENDIF
    '
    IF @tstzone(xm%,ym%,195,138,110,13)
      @sauver_carte
    ENDIF
    '
    IF @tstzone(xm%,ym%,229,14,20,14)
      @aller_haut
    ENDIF
    '
    IF @tstzone(xm%,ym%,251,14,20,14)
      @aller_bas
    ENDIF
    '
    IF @tstzone(xm%,ym%,251,30,20,14)
      @aller_droite
    ENDIF
    '
    IF @tstzone(xm%,ym%,229,30,20,14)
      @aller_gauche
    ENDIF
    '
    IF @tstzone(xm%,ym%,xcarte%,ycarte%,176,176)
      @clic_carte(xm%,ym%)
    ENDIF
    '
    IF @tstzone(xm%,ym%,261,166,14,18)
      @inc_element
    ENDIF
    '
    IF @tstzone(xm%,ym%,225,166,14,18)
      @dec_element
    ENDIF
  LOOP
RETURN
'
' ************************
' *  PROCEDURE PRINCIPALE  *
' ************************
PROCEDURE main
  nb_elements%=0
  ligne_carte%=1
  colonne_carte%=1
  element_courant%=1
  '
  xcarte%=8
  ycarte%=12
  @charge_elements("ELEMENTS.BIN")
  '
  @init_carte
  HIDEM
  GRAPHMODE 2
  @aff_ecran
  @aff_carte
  BMOVE XBIOS(2),adr2%,32000
  '
  @souris_fleche
  SHOWM
  etat%=1
  @gestion_editeur
RETURN
```

Pour finir

Nous verrons le mois prochain comment utiliser les cartes générées par EDITCART pour gérer les déplacements d'une équipe de personnages sur un monde virtuel (affichage de l'image de l'équipe sur la carte, gestion des rencontres, interdictions ou autorisation de déplacements sur certaines cases, etc.) Vous trouverez en téléchargement sur le **3615 ATARI** les listings de l'article, un exemple de carte et une image *Degas* contenant quelques éléments de dessin.

Patrick Leclercq

TOUS LES LISTINGS ET IMAGES DE CE BEST OF GFA BASIC SONT DISPONIBLES EN TELECHARGEMENT SUR LE 3615 ATARI

PROGRAMMATION D'UN JEU DE ROLE (III)

Un jeu de rôle fonctionnel

Voici un petit jeu de rôle type Ultima ou Fantaisy où le joueur se déplace sur une carte de grande dimension et fait de dangereuses rencontres.

Présentation du jeu

Le joueur gère les déplacements d'un personnage dans un univers de type médiéval-fantastique. Cet univers est représenté sous la forme d'une carte de grande dimension dessiné avec des éléments graphiques de base (arbre, maison, montagnes, lacs, etc.) La carte est créée avec le programme EDITCART (voir jeu de rôle II).
La carte est trop grande pour être dessinée entière à l'écran. Le programme en affiche juste une petite portion, représentant les environs immédiats du personnage.
Ce dernier se trouve au milieu de l'image visible. Les déplacements sont réalisés en scrollant l'image de la carte.
L'image de la carte est fabriqué avec 11x11 éléments graphiques mesurant chacun 16x16 pixels. Elle mesure 176x176 pixels.

```
+-------------------------------+
| +---------------+ +----------+ |
| |               | |          | |
| |               | |          | |
| |   Dessin de   | | Zone de  | |
| |   la carte    | | messages | |
| |               | |          | |
| |               | |          | |
| +---------------+ +----------+ |
+-------------------------------+
```

Les fléches du pavé numérique servent à indiquer la direction du déplacement. Le programme interdit le déplacement sur certains types de case (plans d'eau et montagnes).
Lors d'un déplacement, le personnage peut faire des rencontres. Celles-ci peuvent être bénéfiques, neutres ou hostiles. Leur nature dépend du type de case sur laquelle se tient le personnage. La plaine est un endroit assez calme où la majeure partie des rencontres sont neutres. Par contre, la forêt est un endroit dangereux où rôdent toutes sortes de créatures et de monstres. Le programme affiche les rencontres dans une fenêtre graphique située à droite de la carte.
Dans la version actuelle de GESCARTE, toutes les rencontres sont neutres car la place manquait pour écrire un système gérant les combats entre le personnage et les créatures de rencontres. Voir l'article jeu de rôle IV qui présente un système permettant de simuler un combat entre le personnage et une créature quelconque. Vous pourrez fusionner GESCARTE et le système de combat pour obtenir un véritable déplacement avec rencontres et combat.

Programmation de Gescarte

Ce logiciel a été programmé en *GFA Basic 3.5E*, mais il est entièrement compatible avec toutes les versions du *GFA Basic* à partir de la version 3.0. Le code source fait plus de 1000 lignes de programme. Ce programme utilise le graphisme, stocke des données sur disque et lit des données à partir du clavier. Il n'utilise ni la souris, ni les fonctions sonores du ST.

Les données graphiques du programme

Le programme GESCARTE a besoin d'un certain nombre de données graphiques. Comme elles ne peuvent être définies dans le programme, elles sont stockées dans des fichiers. Leur contenu est chargé en mémoire au début du programme. Ces fichiers sont ELEMENTS.BIN, CARTE.CAR et PERSO.BIN.

Le fichier graphique ELEMENTS.BIN

Ce fichier contient les éléments graphiques de 16x16 pixels servant à dessiner l'image de la carte. Ces éléments graphiques sont créées à partir d'une image *Degas Elite*.
La création se fait avec l'utilitaire GENELEM.GFA (voir jeu de rôle II).
La structure du fichier ELEMENT.BIN est la suivante : 32 octets contenant la palette de couleur des éléments, 1 octet contenant le nombre d'éléments du fichier et les éléments graphiques (134 octets par élément). La taille d'une variable alphanumérique contenant un bloc de 16x16 pixels est de 134 octets. Cette taille peut être mesurée avec la fonction LEN.

```
GET 1,1,16,16,bloc$
PRINT LEN(bloc$)
```

La procédure @charge_elements(fichier$) charge dans le tableau alphanumérique element$() les éléments stockés dans le fichier fichier$. Le nombre d'éléments chargés est écrit dans la variable globale nb_elements%.

```
DIM element$(200)
'
PROCEDURE charge_elements(fichier$)
  LOCAL n%
  LOCAL palette$
  '
  palette$=SPACE$(32)
  OPEN "I",#1,fichier$
  BGET #1,VARPTR(palette$),32
  VOID XBIOS(6,L:VARPTR(palette$))
  nb_elements%=INP(#1)
  FOR n%=1 TO nb_elements%
    element$(n%)=SPACE$(134)
    BGET #1,VARPTR(element$(n%)),134
  NEXT n%
  CLOSE #1
RETURN
```

La palette de couleurs est chargée dans une chaîne alphanumérique de 32 caractères. L'instruction SPACE$ permet de créer une variable de taille quelconque. Les 32 octets de la palette sont chargés avec l'instruction BGET.

```
palette$=SPACE$(32)
BGET #1,VARPTR(palette$),32
```

La taille de la chaîne de caractères doit être obligatoirement égale ou supérieure au nombre d'octets chargés en mémoire. Il peut y avoir n'importe quoi stocké en mémoire après l'adresse de la chaîne de caractère. Un mauvais chargement peut écraser ou modifier des variables importantes. C'est une excellente méthode pour créer un "bug furtif" très difficile à trouver.
L'instruction XBIOS(6) permet de remplacer la palette de couleur système par la palette de couleur des éléments graphiques.

```
VOID XBIOS(6,L:VARPTR(palette$))
```

Le nombre d'éléments du fichier est lu grâce à l'instruction INP. Celle-ci permet de lire la valeur d'un octet à partir d'un fichier.

```
nb_elements%=INP(#1)
```

Les éléments sont chargés les uns après les autres. La zone mémoire de 134 octets nécessaire au stockage d'un élément graphique est réservée avec l'instruction SPACE$.

```
FOR n%=1 TO nb_elements%
  element$(n%)=SPACE$(134)
  BGET #1,VARPTR(element$(n%)),134
NEXT n%
```

Les éléments graphiques ont été stockés en mémoire avec l'instruction GET. Pour afficher un élément graphique sur l'écran, il faut utiliser l'instruction PUT. L'exemple suivant affiche l'élément numéro 1 du tableau element$() à la position (10,10).

```
PUT 10,10,element$(1)
```

Le fichier CARTE.CAR

Le fichier CARTE.BIN contient la description de la carte. Il est généré par l'éditeur de carte EDITCART.
La structure de ce fichier est la suivante: un en-tête d'identification de 19 caractères, un octet contenant le nombre de colonnes de la carte, un octet contenant le nombre de lignes de la carte et le codage de la carte proprement dit. Les éléments sont stockés ligne à ligne.
L'en-tête d'identification est la chaîne de caractère "CARTE JdR ATARI MAG". Cette chaîne peut servir à contrôler si le fichier lus contient bien une carte.
La carte est stockée en mémoire dans le tableau carte|(). Ce tableau est du type entier court (entier 8 bits). Etant donné qu'un entier court ne peut prendre que 256 valeurs (de 0 à 255), il ne peut y avoir que 256 éléments possibles.

```
DIM carte|(200,75)
```

Les dimensions du tableau carte|() sont fixés arbitrairement à 200 lignes de 75 éléments. Si vous désirez avoir une carte plus grande, il faut augmenter les dimensions du tableau.
La procédure @charge_carte(nom$) charge en mémoire la carte stockée dans le fichier nom$. Les dimensions de la carte sont écrites dans les variables globales nb_lignes% et nb_colonnes%.

```
PROCEDURE charger_carte(nom$)
  LOCAL entete$
  LOCAL l%,c%
  '
  OPEN "I",#1,nom$
  entete$=SPACE$(19)
  BGET #1,VARPTR(entete$),19
  nb_colonnes%=INP(#1)
  nb_lignes%=INP(#1)
  FOR c%=1 TO nb_colonnes%
    FOR l%=1 TO nb_lignes%
      carte|(c%,l%)=INP(#1)
    NEXT l%
  NEXT c%
  CLOSE #1
RETURN
```

Cette routine charge l'identificateur de fichier, mais ne le teste pas. Si vous désirez le faire, la modification est simple à réaliser, comme le montre l'exemple suivant.

```
entete$=SPACE$(19)
BGET #1,VARPTR(entete$),19
IF entete$<>"CARTE JdR ATARI MAG".
  PRINT "Ce fichier ne contient pas de carte"
ELSE
```

```
  ' Chargement carte
  ...
ENDIF
```

Le fichier graphique PERSO.BIN

Le fichier PERSO.BIN contient l'image du personnage de 16x16 pixels. Cette image est affichée sur la carte pour visualiser la position du personnage. La procédure @charge_perso(nom$) charge en mémoire l'image stockée dans le fichier nom$. L'image est mémorisée dans la variable globale perso$.
L'image doit être affichée en superposition sur la carte. L'opérateur graphique logique AND permet d'afficher une image en superposition sur un décor quelconque à condition que l'image soit dessinée en couleur 0 sur un fond de couleur 15.

```
PUT px,py,perso$,1
```

Cette technique d'affichage est un "affichage en ombre chinoise". Pour obtenir plus de détails sur les opérateurs graphiques logiques et leurs utilisations, vous pouvez vous référer à l'article: «Opérateur graphique logique».
L'image PERSO.BIN est fabriqué par le programme IMGPERSO.GFA à partir d'une image *Degas* contenant la silhouette du personnage.

```
' ********************************
' *  SAISIE IMAGE DU PERSONNAGE  *
' ********************************
RESERVE 100000
'
@load_degas("ELEMENTS.PI1")
GET 1,35,16,50,perso$
OPEN "o",#1,"perso.bin"
BPUT #1,VARPTR(perso$),134
CLOSE #1
END
'
' **********************
' *  CHARGEMENT IMAGE  *
' *     DEGAS ELITE    *
' **********************
PROCEDURE load_degas(nom$)
  LOCAL palette$
  '
  palette$=SPACE$(32)
  OPEN "I",#1,nom$
  SEEK #1,2
  BGET #1,VARPTR(palette$),32
  VOID XBIOS(6,L:VARPTR(palette$))
  BGET #1,XBIOS(2),32000
  CLOSE #1
RETURN
```

Structure de Gescarte

Le programme GESCARTE est constitué d'un certain nombre de modules. Un module est un ensemble de routines qui traite un problème donné. Par exemple, le module de déplacement s'occupe de gérer entièrement le déplacement sur la carte (affichage de l'image de la carte, mise à jour des variables internes, tests des déplacements possibles, appel du module de rencontre, etc.) Un module est composé d'une ou plusieurs routines principales et de sous-routines dépendant des routines principales.
Les modules de GESCARTE sont: initialisation, module principal, interface utilisateur, déplacement sur la carte, gestion des rencontres et gestion de la fenêtre graphique. Il y a aussi un module de fonctions systèmes. Les fonctions systèmes sont des procédures de base qui peuvent être réutilisées dans n'importe quel type de programme.

Module d'initialisation

Il est constitué par la procédure @main, qui initialise les variables du programme, charge en mémoire les données du jeu et affiche l'image de la carte.
Les données graphiques sont stockées sur disque et les données numériques sont définis dans des lignes de DATA. L'initialisation de chaque type de données est réalisé par une procédure particulière appartenant à divers modules. Par exemple, la procédure @init_rencontres appartient au module de gestion des rencontres.
Une fois l'initialisation terminée, la procédure @main appelle la procédure @gestion_programme. Cette procédure est la routine principale du module principale de GESCARTE.

Module interface utilisateur

L'interface utilisateur est le module qui s'occupe de la communication entre le joueur et le programme. Il est très important. Une bonne interface utilisateur peut faire la différence entre un bon programme agréable à utiliser et un mauvais programme peu pratique à utiliser. Plusieurs logiciels très performants sur le plan technique ont été des échecs commerciaux à cause d'une mauvaise interface utilisateur.
L'interface utilisateur de GESCARTE est assurée par les routines @att_ev, @quitter_programme et le module de gestion de la fenêtre graphique.
La fonction @att_ev attend que l'utilisateur presse sur une fléche du pavé numérique ou sur la touche [Esc] pour renvoyer un code de fonctions spécifique au programme.

```
1 : Fléche haute
2 : Fléche bas
3 : Fléche droite
4 : Fléche gauche
5 : Touche [Esc]
```

Si vous décidez de modifier l'interface utilisateur de GESCARTE, il suffit de modifier @att_ev sans changer le reste du programme. C'est l'avantage d'une communication inter-routines par codes numériques. Pour rajouter de nouvelles fonctions au programme (examen des possessions du personnage, sommeil, etc.), il faut définir l'action correspondant à la nouvelle fonction (touche du clavier, clic souris, etc.) et lui donner un code. Tous les codes supérieurs à 6 sont disponibles.
La procédure @quitter_programme demande à l'utilisateur une confirmation de sa décision de quitter le programme. Elle affiche une

boîte d'alerte avec le message "Voulez-vous vraiment quitter ce prg?". Si le joueur confirme sa décision, la variable globale sortie% est mise à 1.

Module principal

Le module principal gère l'exécution du programme.
Il est constitué de la procédure @gestion_programme et de la procédure @exec_fonction.

```
PROCEDURE gestion_programme
  LOCAL ev%
  '
  DO
    ev%=@att_ev
    @exec_fonction(ev%)
    EXIT IF sortie%=1
  LOOP
RETURN
```

La procédure @gestion_programme contient la boucle principale du programme. Cette boucle s'exécute tant que la variable globale sortie% ne contient pas 1. La fonction @att_ev lui donne le code de la commande sélectionnée par le joueur.

```
PROCEDURE exec_fonction(f%)
  SELECT f%
  CASE 1,2,3,4
    @gestion_deplacement(f%)
  CASE 5
    @quitter_programme
  ENDSELECT
RETURN
```

La fonction @exec_fonction(ev%) exécute la routine associée à la fonction utilisateur f%. C'est un système d'aiguillage qui dirige le déroulement du programme vers les routines de déplacement ou vers la routine gérant la sortie du programme. Si vous rajoutez des fonctions à votre programme, c'est ici qu'il faut implanter les appels des nouvelles procédures.
Par exemple, une fonction permettant d'examiner les possessions du personnage (numéro 6), ou une fonction permettant de changer l'arme du personnage (numéro 7), ou de boire une potion (numéro 8), etc.

```
SELECT f%
CASE 1,2,3,4
  @gestion_deplacement(f%)
CASE 5
  @quitter_programme
CASE 6
  @examen_possessions
CASE 7
  @changer_arme
CASE 8
  @boire_potion
ENDSELECT
```

Module de déplacement sur la carte

Il gère les déplacements sur la carte. C'est le module le plus complexe du programme.
Les fonctions appartenant à ce module sont: @gestion_deplacement, @init_evenements, @exec_deplacement, @aller_droite, @aller_gauche, @aller_haut, @aller_bas, @preparation_ecr_travail et @visualisation_ecr_travail.
Avant de plonger dans les détails techniques, voyons comment s'effectuent les déplacements. Théoriquement, le personnage se déplace de case en case. Ce serait simple à gérer si toutes les cases était les mêmes, ce qui n'est pas le cas. Le déplacement est possible sur la plupart des cases, mais pas toutes (plan d'eau par exemple). Le personnage peut aussi rencontrer des créatures sur certaines cases.
Le programme associe un numéro à chaque type de traitement. Le numéro 0 correspond à une impossibilité de déplacement. Le numéro 1 correspond à un déplacement simple. Le numéro 2 correspond à un déplacement sur une plaine. le numéro 3 correspond à un déplacement en forêt et le numéro 4 à un déplacement sur des ruines. Le tableau dev_ev%(num_element%) contient le numéro de traitement associé à l'élément graphique num_element%. Les associations entre les numéros d'éléments graphiques et les traitements sont définies dans des lignes de DATA. La procédure @init_evenements initialise le tableau dev_ev%() avec les données des DATA.

```
def_evenements:
DATA 1,0
DATA 2,0
DATA 3,2
DATA 4,2
DATA 5,0
DATA 6,0
DATA 7,0
DATA 8,0
DATA 9,4
DATA 10,0
DATA 13,3
DATA -1
```

Les éléments graphiques 1,2,5,6,7,8 et 9 (plan d'eau, montagne, maisons, villages et château) ont le numéro de traitement 0, c'est-à-dire que le personnage ne peut se déplacer sur une case de la carte occupé par l'un de ces éléments.
Le traitement 2 (déplacement sur plaine avec possibilité de rencontre) correspond aux éléments graphiques 3 et 4 (dessins de plaines). Lorsque le programme déplace le personnage sur une case de la carte occupée par un élément de plaine, il appelle la routine gérant les rencontres en plaine (procédure @rencontre_plaine).
Le traitement 3 (déplacement en forêt avec possibilité de rencontre) correspond à l'élément graphique 13 (dessin de forêt). Le traitement 4 (déplacement sur ruines avec possibilité de rencontre) correspond à l'élément graphique 9 (dessin de ruines).

Routine @gestion_deplacement

La procédure @gestion_deplacement(direction%) sélectionne le type de déplacement en fonction du type d'élément sur lequel le per-

sonnage vas se déplacer. Une structure SELECT-ENDSELECT sélectionne les routines à appeler en fonction du type de traitement.

```
' La case où veut se rendre
' le personnage est définis
' par la position (c%,l%).
'
n%=dev_ev%(carte|(c%,l%))
'
SELECT n%
CASE 1
  @exec_deplacement(direction%)
CASE 2
  @exec_deplacement(direction%)
  @rencontre_plaine
CASE 3
  @exec_deplacement(direction%)
  @rencontre_foret
CASE 4
  @exec_deplacement(direction%)
  @rencontres_ruines
ENDSELECT
```

Limitations aux bords de la carte

Le programme ne gère pas les déplacements sur les bords de l'écran. Rien n'empêche donc le personnage de dépasser les limites de la carte. Pour éviter cela, il faut placer une barrière d'éléments infranchissables aux limites de la carte. Ce peut être de l'eau ou des montagnes. De toute façon, les bords de la carte doivent être visualisés d'une manière logique pour le joueur. De nombreux programmes du commerce sont limités par la mer. La carte fournie en téléchargement sur le serveur **3615 ATARI** avec les programmes EDITCART et GESCARTE (sous le nom de JEUGFA24.TOS) est limitée par une vaste étendue d'eau sur laquelle le joueur ne peut se déplacer.

Routine @exec__deplacement

La routine @exec_deplacement(direction%) exécute un déplacement simple dans la direction direction%. C'est une fonction d'aiguillage qui sélectionne la routine de déplacement à utiliser pour gérer le déplacement dans la direction désiré.

```
PROCEDURE exec_deplacement(direction%)
  SELECT direction%
  CASE 1
    @aller_haut
  CASE 2
    @aller_bas
  CASE 3
    @aller_droite
  CASE 4
    @aller_gauche
  ENDSELECT
RETURN
```

Les procédures @aller_haut, @aller_bas, @aller_droite et @aller_gauche déplacent l'image de la carte dans une direction précise. Leurs structures est assez complexe car redessiner rapidement l'image de la carte avec des routines écrites entièrement en *GFA Basic* n'est pas simple.

Il est impossible de redessiner l'image de la carte sur l'écran suffisamment rapidement pour que le joueur ne voie pas l'opération s'effectuer. Ce serait possible en *Assembleur*, mais pas en *GFA Basic*. Pour éviter cet effet graphique désagréable à l'oeil, les routines de déplacements effectuent les opérations graphiques dans un écran virtuel situé en mémoire.

Une fois les opérations graphiques terminés, le programme devrait recopier l'image de la carte sur l'écran, mais l'expérience prouve que cette technique n'est pas assez rapide pour obtenir un affichage de bonne qualité. L'image flashe un peu. Pour éviter cela, le programme ne recopie pas l'écran virtuel dans la mémoire écran du ST, mais change l'adresse de la mémoire écran par l'adresse de la mémoire virtuelle. L'écran virtuel devient alors la nouvelle mémoire écran.

Le programme travaille sur deux zones mémoires qui sont alternativement utilisés comme mémoire écran et écran virtuel. Leurs rôles sont inversés à chaque affichage. Les adresses de ces zones mémoire sont stockées dans les variables adr1% et adr2%. L'adresse de l'écran visible à l'écran est stockée dans la variable ecr_visible%. L'adresse de l'écran virtuel est stockée dans la variable ecr_travail%.

La position de la portion de carte visible par rapport à la carte totale est stockée dans les variables ligne_carte% et colonne_carte%. Les routines de déplacement modifient ces variables en fonction du sens de déplacement.

La routine @aller__bas

Pour comprendre comment fonctionnent ces routines, la meilleure méthode est encore de prendre un exemple.

Ce sera la routine @aller_bas qui déplace l'image de la carte vers le bas. L'image de la carte est constituée d'un ensemble de 11x11 éléments graphiques, c'est-à-dire de 11 lignes constituées chacune de 11 éléments graphiques. L'image du personnage est affichée sur la case située au milieu de la carte (sixième ligne, sixième colonne).

Le déplacement ne s'effectue que d'une ligne à la fois. Seule une partie de l'image est modifiée. Les lignes 2 à 11 sont toujours affichées sur l'écran, avec un décalage d'une ligne vers le haut. La ligne 2 se retrouve à la place de la ligne 1.

La ligne 3 se retrouve à la place de la ligne 2, etc. Seule la ligne 11 doit être redessinée par le programme. Cette technique de réaffichage partiel permet de gagner beaucoup de temps par rapport à un réaffichage complet.

La procédure @aller_bas recopie dans l'écran virtuel les lignes 2 à 11 à la position des lignes 1 à 10. Ensuite, elle redessine l'élément du milieu de la carte qui avait été altéré par l'affichage de l'image du personnage. Elle dessine ensuite la nouvelle ligne, puis affiche l'image du personnage au milieu de la carte. Une fois la carte terminée, le programme change l'adresse de la mémoire écran par l'adresse de l'écran virtuel. Le changement de l'adresse écran est effectué par la procédure @visualisation_ecr_travail.

Etant donné que les adresses de la mémoire écran et de l'écran virtuel sont inversés lors des déplacements, les messages qui sont affichés dans la fenêtre graphique disparaissent à chaque déplacement pour réapparaître au suivant. Pour éviter cela, la procédure @co-

pie_fenetre recopie le contenu de la fenêtre graphique dans l'écran virtuel juste avant l'inversion des adresses.
Remarque: La procédure @visualisation_ecr_travail change l'adresse de la mémoire écran par l'adresse de l'écran virtuel (ecr_travail%). Cette procédure utilise la procédure @set_ecr_phys qui utilise elle-même la fonction XBIOS(5). Cette instruction système n'a pas une action instantanée : elle attend que le spot vidéo ait finit de dessiner l'image pour changer l'adresse de la mémoire écran. L'instruction VSYNC bloque le déroulement du programme tant que le spot vidéo dessine l'écran, et donc tant que la fonction XBIOS(5) n'a pas changée l'adresse de la mémoire écran. Cela permet d'éviter des "bavures vidéo" désagréables à l'œil.

```
PROCEDURE visualisation_ecr_travail
  @set_ecr_phys(ecr_travail%)
  VSYNC
RETURN
```

Module de rencontres

Le module de rencontre gère les rencontres et les découvertes que peut faire le joueur en se déplaçant sur la carte.
Il est constitué des routines @rencontre_plaine, @rencontre_foret, @rencontre_ruines, @tst_probabilite, @tirage_rencontre, @init_rencontres, @tirage_decouverte, @init_decouvertes et @init_tables_decouvertes.
Toutes les cases où peuvent apparaître des créatures ont une probabilité de rencontre. Lorsque le personnage entre dans l'une de ces cases, le programme effectue un tirage de probabilité pour déterminer s'il se produit effectivement une rencontre.
La fonction @tst_probabilite(prob%) permet de savoir si un événement de probabilité prob% se produit effectivement. Elle renvoie la valeur logique vraie (TRUE) si l'événement se produit et la valeur logique fausse (FALSE) si l'événement ne se produit pas. Le tirage d'une probabilité comprise entre 1 et 100 se fait avec l'instruction RANDOM.
Une fois que le programme sait qu'il se produit une rencontre, il détermine la nature de la créature en utilisant une table de rencontre. Une table de rencontre est une liste de créatures possédant chacune une probabilité d'apparition.
La fonction @tirage_creature(table%) tire une créature de la table de rencontre table%. Il existe une table de rencontre pour la plaine, une pour la forêt et une pour les ruines.
Les routines @rencontre_plaine, @rencontre_foret et @rencontre_ruines gérent les 3 types de rencontres qui peuvent se produire dans GESCARTE Pour comprendre comment fonctionnent ces routines, prenons l'exemple de @rencontre_plaine. Cette procédure gère les rencontres que le personnage peut faire dans un environnement de plaine.
La probabilité de rencontre est de 10%. Si la rencontre se produit, la fonction @tirage_creature permet de déterminer le type de créature rencontrée. Le programme affiche alors le nom de la créature dans la fenêtre graphique. L'affichage se fait avec la procédure @aff_fenetre.

```
PROCEDURE rencontre_plaine
  LOCAL creature%
  LOCAL m$
  '
  IF @tst_probabilite(10)
    creature%=@tirage_creature(1)
    m$="RENCONTRE : "+creature$(creature%)
    @aff_fenetre(m$)
  ENDIF
RETURN
```

La routine @rencontre_ruines gère les rencontres, mais aussi la découverte d'un trésor. La gestion d'une découverte se fait de la même manière que la gestion d'une rencontre: test de probabilité et table de découverte.

Initialisation des tables de rencontres

Une table de rencontre est une liste de créatures avec une probabilité de rencontre. La somme des probabilités d'une table doit être de 100. Le tableau table_rencontre%() contient les différentes tables de rencontre du programme.
Il peut mémoriser 20 tables pouvant contenir chacune jusqu'à 30 probabilités de rencontre.

```
DIM table_rencontre%(20,30,2)
```

Le tableau table_rencontre%(t%,r%,1) contient la probabilité de rencontre avec la créature table_recontre%(t%,r%,2).
Les tables sont définies dans des lignes de DATA sous la forme numéro créature/probabilité de rencontre. La valeur -2 signifie qu'il y a encore une table après la table courante. la valeur -1 est l'indicateur de fin de données. La table 1 est la table de rencontre en plaine. la table 2 est la table de rencontre en forêt et la table 3 est la table de rencontre des ruines.

```
' **********************
' *      TABLE 1       *
' *      PLAINE        *
' **********************
' *  Paysans    : 50%  *
' *  Colporteur : 10%  *
' *  Voleurs    : 20%  *
' *  Chiens     : 10%  *
' *  Orcs       : 10%  *
' **********************
DATA 21,50
DATA 22,10
DATA 20,20
DATA 2,10
DATA 50,10
DATA -2
```

Définition des découvertes

Les objets que le personnage peut découvrir sont définis par une description stockée dans le tableau alphanumérique decouverte$().
La description de ces objets est définis dans des lignes de DATA.
La procédure @init_decouvertes initialise le tableau decouverte$().

Définition des créatures

Les créatures que le personnage peut rencontrer sont définies uniquement par leurs noms. Ceux-ci sont stoçkés dans le tableau creature$(). Les créatures sont définies dans des lignes de DATA. La procédure @init_creatures initialise le tableau creature$() avec les données stockées en DATA. Les créatures sont définies de manière "minimale". Si vous voulez réalisez un système de combat, il faut rajouter d'autres caractéristiques.

Module de gestion de la fenêtre graphique

La fenêtre graphique est une zone située à la droite de la carte où le programme peut afficher des messages. Le programme GESCARTE l'utilise pour afficher les rencontres et les découvertes de trésors.
Elle est située à la position (194,12) et mesure 118 pixels de large et 178 pixels de haut. Elle peut contenir 20 lignes de 14 caractères. La variable globale ligne_courante% contient la position courante d'affichage.
La procédure @cls_fenetre dessine l'image de la fenêtre vide. Elle sert à afficher la fenêtre graphique au début du programme et à l'effacer en cours de route.
La procédure @init_fenetre affiche l'image de la fenêtre et met la variable ligne_courante% à 0.
La procédure @aff_fenetre(message$) affiche le texte message$ dans la fenêtre graphique. Le texte est mis en forme par la routine @format pour ne pas dépasser les limites de la fenêtre. Chaque ligne formatée est affichée par la fonction @aff_ligne.
La routine @aff_ligne(ligne$) affiche la ligne ligne$ à la position courante de la fenêtre. La ligne est affichée à la position stockée dans la variable ligne_courante%. Si celle-ci contient une valeur supérieure à 19, il ne reste plus de place libre dans la fenêtre car les numéros de lignes sont comprises entre 0 et 19. Le programme scrolle alors la fenêtre graphique d'une ligne vers le haut.
La procédure @scroll_fenetre scrolle le contenu de la fenêtre graphique d'une ligne vers le haut. Le scrolling est réalisé en copiant les lignes 1 à 19 à l'emplacement des lignes 0 à 18. La copie est faite avec les instructions graphiques GET et PUT. Une fois le scrolling effectué, la dernière ligne est effacée.
La procédure @copie_fenetre recopie le contenu de la fenêtre graphique de l'écran physique vers l'écran logique (écran visible vers écran de travail). Cette procédure est utilisée par le système d'affichage de la carte qui travaille en permutant les écrans physique et logique.

Les fonctions systémes

Les fonctions systèmes sont des fonctions multi-usages qui remplissent des tâches classiques dans toutes sortes de programmes. Ils ne dépendent pas de GESCARTE et peuvent être réutilisés dans toutes sortes de programmes.
Les fonctions systèmes sont @waitclavier, @vider_clavier, @cls_ecran, @cadre, @set_ecr_log, @set_ecr_phys et @format
La procédure @waitclavier attend que l'utilisateur presse sur une touche. Elle ne renvoie pas le code de la touche pressée. C'est l'équivalent de l'instruction VOID INP(2). A ceci prés qu'il est plus parlant de trouver @waitclavier dans un listing que VOID INP(2).
La procédure @vider_clavier efface tous les caractères stockés dans le tampon clavier. Elle permet d'annuler l'effet de la fonction d'auto-répétition du clavier ST. Cela peut être utile dans certains cas.
La routine @cls_ecran(c%) efface l'écran en le remplissant avec la couleur c%. Cette routine est plus pratique que l'instruction CLS qui remplis uniquement l'écran avec la couleur 0. Cela permet d'avoir un fond d'écran de n'importe quelle couleur.
La procédure @cadre(px%,py%,tx%,ty%,f%,b%) dessine un cadre graphique de dimensions (tx%,ty%) à la position (px%,py%). Le fond du cadre est de la couleur f%, alors que les bords sont dans la couleur b%. C'est une sorte d'instruction PBOX amélioré.
La routine @set_ecr_log(adr_ecran%) permet de changer l'adresse de l'écran logique, c'est-à-dire l'adresse de l'écran où travaillent les fonctions graphiques du GFA Basic. Le nouvel écran de travail doit faire 32000 octets pour éviter les plantages systèmes. Cette routine permet de réaliser des opérations graphiques en mémoire sans que rien n'apparaisse à l'écran.
La procédure @set_ecr_phys(adr_ecran%) permet de changer l'adresse de l'écran physique, c'est-à-dire l'adresse de la mémoire écran. C'est une fonction dangereuse qu'il faut employer avec prudence. La nouvelle mémoire écran doit faire 32000 octets et commencer à une adresse multiple de 256 (cette limitation n'existe pas sur le STE où l'adresse de la mémoire écran peut être fixée à n'importe quelle adresse paire). En utilisant cette fonction avec @set_ecr_log, on peut réaliser des affichages très rapides, mais assez complexes.
La routine @format(m$,taille%) est une routine qui formate le texte m$ en lignes ne dépassant pas taille% caractères. Le texte est formaté de manière à ne pas couper les mots. Le texte formaté est stocké dans le tableau alphanumérique lignes$(). La variable globale nb_lignes% contient le nombre de lignes formatées présentes dans le tableau lignes$().

Améliorations du programme

Le programme GESCARTE peut être amélioré de nombreuses manières. Vous pouvez ajouter de nouveaux éléments graphiques et de nouvelles tables de rencontres. Un système de sous-cartes peut permettre au personnage d'entrer dans les maisons et d'explorer les châteaux.
L'interface utilisateur peut être modifiée pour fonctionner entièrement à la souris. Des icônes peuvent représenter les différents options du programme (déplacement, examen du personnage, sommeil, manipulations des objets possédés par le personnage, sauvegarde de la partie, chargement d'une ancienne partie, etc.)
Vous pouvez prévoir une possibilité de dialogues entre le personnage et les créatures qu'il rencontre pendant ses déplacements.
Pour transformer ce logiciel en véritable jeu de rôle micro, il faut prévoir un but et des sous-buts. En général, ce but est la destruction d'un puissant sorcier. Les sous-buts peuvent être la recherche d'informations, d'objets magiques très puissants (armes, armures et objets divers) et éventuellement de sorts. Certains endroits de la carte peuvent être extrêmement dangereux pour un personnage débutant sans expérience et sans équipement particulier.

Téléchargement

Vous pouvez obtenir les listings de EDITCART et de GESCARTE sur le

3615 ATARI. Ces programmes sont accompagnés d'images *Degas Elite* contenant des exemples d'éléments graphiques de 16x16 pixels et un exemple de carte.

```
' ********************************
' *   JEU DE ROLE STYLE ULTIMA   *
' *         GFA BASIC 3.xx       *
' ********************************
' *  (C) 1991 ATARI MAGAZINE     *
' *  (C) 1991 PATRICK LECLERCQ   *
' ********************************
'
RESERVE 100000
' OPTION BASE 1
DIM element$(200)
'
DIM creature$(200)
'
DIM decouverte$(200)
'
DIM lignes$(20)
'
DIM table_rencontre%(20,30,2)
DIM table_decouverte%(20,30,2)
'
DIM def_ev%(200)
'
adr_tampon%=MALLOC(32255)
adr2%=(adr_tampon%+255) AND &HFFFFFF00
adr1%=XBIOS(2)
'
nb_colonnes%=-1
nb_lignes%=-1
DIM carte|(200,75)
'
@main
'
' ************************************
' *  RETOUR AUX CONDITIONS DE DEBUT  *
' ************************************
@set_ecr_log(adr1%)
@set_ecr_phys(adr1%)
CLS
VOID MFREE(adr_tampon%)
END
'
' ____________________________________
' ************************************
' *  ATTENTE PRESSION SUR UNE TOUCHE *
' ************************************
' *  Cette fonction ne renvoie pas   *
' *  le numéro de touche. C'est      *
' *  l'équivalent de l'instruction   *
' *  VOID INP(2)                     *
' ************************************
PROCEDURE waitclavier
  DO
    EXIT IF INKEY$<>""
  LOOP
RETURN
'
' **************************
' *  VIDER TAMPON CLAVIER  *
' **************************
PROCEDURE vider_clavier
  DO
    EXIT IF INKEY$=""
  LOOP
RETURN
'
' ____________________________
PROCEDURE set_ecr_log(adr_ecran%)
  VOID XBIOS(5,L:adr_ecran%,L:-1,-1)
RETURN
'
PROCEDURE set_ecr_phys(adr_ecran%)
  VOID XBIOS(5,L:-1,L:adr_ecran%,-1)
RETURN
'
' **********************
' *  EFFACEMENT ECRAN  *
' **********************
' ******************************************
' *  c% : couleur de remplissage de l'écran *
' ******************************************
PROCEDURE cls_ecran(c%)
  DEFFILL c%
  PBOX 0,0,319,199
RETURN
'
'
' ******************************
' *  AFFICHAGE CADRE GRAPHIQUE *
' ******************************
PROCEDURE cadre(px%,py%,tx%,ty%,f%,b%)
  LOCAL px2%,py2%
  '
  px2%=px%+tx%-1
  py2%=py%+ty%-1
  DEFFILL f%
  PBOX px%,py%,px2%,py2%
  COLOR b%
  BOX px%,py%,px2%,py2%
RETURN
'
'
' ____________________________
' ****************************
' *                          *
' *  ROUTINES DE GESTION DE  *
' *  LA FENTRE DES MESSAGES  *
' *                          *
' ****************************
```

```
' ____________________
' *************************
' *  FORMATAGE D'UN TEXTE  *
' *************************
'
*******************************************************
' * Cette routine découpe un texte en plusieurs lignes*
' * pour un affichage propre. Le découpage se fait au *
' * niveau des mots.                                  *
' * PARAMETRES ROUTINE :                              *
' * ___________________                               *
' * m$      : texte à formater                        *
' * taille% : nombre maximum de lettres par ligne     *
' * SORTIE ROUTINE :                                  *
' * ________________                                  *
' * Les lignes sont stockées dans le tableau lignes$()*
' * La variable nb_lignes% contient le nb de lignes du*
' * tableau lignes$().                                *
' *******************************************************
PROCEDURE format(m$,taille%)
  LOCAL tampon$
  LOCAL p%
  LOCAL c$
  LOCAL ligne$
  '
  nb_lignes%=0
  tampon$=""
  ligne$=""
  '
  FOR p%=1 TO 20
    lignes$(p%)=""
  NEXT p%
  '
  ' *************************
  ' *  BOUCLE DE DECOUPAGE  *
  ' *************************
  FOR p%=1 TO LEN(m$)
    c$=MID$(m$,p%,1)
    '
    IF c$<>" "
      tampon$=tampon$+c$
    ENDIF
    '
    IF c$=" "
      IF LEN(ligne$)+1+LEN(tampon$)<=taille%
        IF ligne$=""
          ligne$=tampon$
          tampon$=""
        ELSE
          ligne$=ligne$+" "+tampon$
          tampon$=""
        ENDIF
      ELSE
        INC nb_lignes%
        lignes$(nb_lignes%)=ligne$
        ligne$=""
        IF tampon$<>""
          tampon$=tampon$+" "
        ENDIF
      ENDIF
    ENDIF
    '
  NEXT p%
  '
  ' **********************
  ' *  FIN DU DECOUPAGE  *
  ' **********************
  IF (ligne$="") AND (tampon$<>"")
    INC nb_lignes%
    lignes$(nb_lignes%)=tampon$
    tampon$=""
  ENDIF
  '
  IF (tampon$<>"") AND (ligne$<>"")
    IF LEN(ligne$)+1+LEN(tampon$)<=taille%
      INC nb_lignes%
      lignes$(nb_lignes%)=ligne$+" "+tampon$
      ligne$=""
      tampon$=""
    ELSE
      INC nb_lignes%
      lignes$(nb_lignes%)=ligne$
      ligne$=""
      INC nb_lignes%
      lignes$(nb_lignes%)=tampon$
      tampon$=""
    ENDIF
  ENDIF
  '
RETURN
'
' ********************************
' *  EFFACEMENT FENETRE MESSAGES  *
' ********************************
PROCEDURE cls_fenetre
  @cadre(194,12,118,178,5,0)
RETURN
'
' ****************************
' *  SCROLLING VERS LE HAUT  *
' ****************************
PROCEDURE scroll_fenetre
  LOCAL scroll$
  '
  GET 196,24,308,185,scroll$
  PUT 196,15,scroll$
  @cadre(196,177,112,9,5,5)
  ! Effacement dernière ligne
RETURN
'
' *****************************************
' *  AFFICHAGE D'UNE LIGNE DANS LA FENETRE  *
' *****************************************
PROCEDURE aff_ligne(ligne$)
```

```
  LOCAL px%,py%
  '
  IF ligne_courante%>19
    @scroll_fenetre
    ligne_courante%=19
  ENDIF
  px%=196
  py%=(ligne_courante%-1)*9+23-1
  TEXT px%,py%,ligne$
RETURN
'
' *******************************************
' *  AFFICHAGE D'UN MESSAGE DANS LA FENETRE  *
' *******************************************
PROCEDURE aff_fenetre(message$)
  LOCAL l%
  '
  @format(message$,14)
  FOR l%=1 TO nb_lignes%
    INC ligne_courante%
    @aff_ligne(lignes$(l%))
  NEXT l%
RETURN
'
' ***********************************
' *  INITIALISATION FENETRE TEXTE  *
' ***********************************
PROCEDURE init_fenetre
  @cls_fenetre
  ligne_courante%=0
RETURN
'
' ****************************************************
' *  COPIE FENETRE DE TEXTES DANS ECRAN DE TRAVAIL  *
' ****************************************************
PROCEDURE copie_fenetre
  RC_COPY ecr_visible%,194,12,118,178
          TO ecr_travail%,194,12
RETURN
'
'
' ____________________________________
' ***********************************
' *                                 *
' *  CHARGEMENT DE FICHIERS DISQUE  *
' *                                 *
' ***********************************
' ____________________________________
' ************************************************
' *  CHARGEMENT IMAGE DU PERSONNAGE SUR LA CARTE  *
' ************************************************
PROCEDURE charge_perso(nom$)
  perso$=SPACE$(134)
  OPEN "i",#1,nom$
  BGET #1,VARPTR(perso$),134
  CLOSE #1
RETURN
'
' ***************************************
' *  CHARGEMENT DU FICHIER BINAIRE      *
' *  CONTENANT LES ELEMENTS GRAPHIQUES  *
' ***************************************
PROCEDURE charge_elements(fichier$)
  LOCAL n%
  LOCAL palette$
  '
  palette$=SPACE$(32)
  OPEN "I",#1,fichier$
  BGET #1,VARPTR(palette$),32
  VOID XBIOS(6;L:VARPTR(palette$))
  nb_elements%=INP(#1)
  FOR n%=1 TO nb_elements%
    element$(n%)=SPACE$(134)
    BGET #1,VARPTR(element$(n%)),134
  NEXT n%
  CLOSE #1
RETURN
'
' *************************************
' *  CHARGER CARTE A PARTIR DU DISQUE  *
' *************************************
PROCEDURE charger_carte(nom$)
  LOCAL entete$
  LOCAL l%,c%
  '
  OPEN "I",#1,nom$
  entete$=SPACE$(19)
  BGET #1,VARPTR(entete$),19
  nb_colonnes%=INP(#1)
  nb_lignes%=INP(#1)
  FOR c%=1 TO nb_colonnes%
    FOR l%=1 TO nb_lignes%
      carte|(c%,l%)=INP(#1)
    NEXT l%
  NEXT c%
  CLOSE #1
RETURN
'
'
' ____________________________
' ****************************
' *  OPTION 'QUITTER LE PRG'  *
' ****************************
' ****************************************
' *  sortie%=0 ==> continuer programme  *
' *  sortie%=1 ==> quitter programme    *
' ****************************************
PROCEDURE quitter_programme
  LOCAL m$,t$,b%
  '
  sortie%=0
  m$="VOULEZ-VOUS VRAIMENT  |QUITTER CE PRG ?"
  choix$="OUI|NON"
  SHOWM
```

```
  ALERT 0,m$,2,choix$,b%
  HIDEM
  IF b%=1
    sortie%=1
  ENDIF
RETURN
'
' ____________________________
' ******************************
' *                            *
' *  GESTIONNAIRE DE RENCONTRES  *
' *                            *
' ******************************
' ********************************************
' *  INITIALISATION DES TABLES DE RENCONTRES  *
' ********************************************
PROCEDURE init_rencontres
  LOCAL table%,rencontre%
  LOCAL prob%,creature%
  LOCAL valeur%
  '
  ARRAYFILL table_rencontre%(),-1
  table%=1
  rencontre%=1
  RESTORE def_rencontres
  DO
    DO
      READ valeur%
      EXIT IF valeur%=-1
      EXIT IF valeur%=-2
      creature%=valeur%
      READ prob%
      table_rencontre%(table%,rencontre%,1)=prob%
      table_rencontre%(table%,rencontre%,2)=creature%
      INC rencontre%
    LOOP
    EXIT IF valeur%=-1
    INC table%
    rencontre%=1
  LOOP
RETURN
'
' ************************
' *  TEST DE PROBABILITE *
' ************************
' ****************************************************
' *  Cette fonction sert à tester si un événement    *
' *  de probabilité prob% s'est produit. La routine  *
' *  renvoie TRUE si l'événement s'est produit et    *
' *  FALSE dans le cas contraire.                    *
' ****************************************************
FUNCTION tst_probabilite(prob%)
  LOCAL rep%
  '
  IF RANDOM(100)+1<=prob%
    rep%=TRUE
  ELSE
    rep%=FALSE
  ENDIF
  RETURN rep%
ENDFUNC
'
' ****************************
' *  TIRAGE D'UNE RENCONTRE  *
' ****************************
' ****************************************************
' * Cette routine détermine aléatoirement un numéro de
' * créature à partir de la table de rencontre table%.
' ****************************************************
FUNCTION tirage_rencontre(table%)
  LOCAL tirage%,rencontre%
  LOCAL prob%,creature%
  LOCAL total_prob%
  '
  tirage%=RANDOM(100)+1
  creature%=-1
  total_prob%=0
  FOR rencontre%=1 TO 30
    prob%=table_rencontre%(table%,rencontre%,1)
    IF prob%<>-1
      ADD total_prob%,prob%
      IF (tirage%<=total_prob%)
        creature%=table_rencontre%(table%,rencontre%,2)
        rencontre%=30  ! Sortie de la boucle
      ENDIF
    ENDIF
  NEXT rencontre%
  '
  IF creature%=-1
    PRINT "ERREUR DE FORMAT DANS LES RENCONTRES"
    PRINT "NUMERO TABLE ";table%
  ENDIF
  RETURN creature%
ENDFUNC
'
' ____________________________
' ********************************
' *                              *
' *  GESTIONNAIRE DES DECOUVERTES  *
' *                              *
' ********************************
' ********************************************
' *  INITIALISATION DES TABLES DE DECOUVERTES  *
' ********************************************
PROCEDURE init_tables_decouvertes
  LOCAL table%,decouverte%
  LOCAL prob%,tresor
  LOCAL info%
  '
  ARRAYFILL table_decouverte%(),-1
  table%=1
  decouverte%=1
  RESTORE tables_decouvertes
```

```
  DO
    DO
      READ info%
      EXIT IF info%=-1
      EXIT IF info%=-2
      tresor%=info%
      READ prob%
      table_decouverte%(table%,decouverte%,1)=prob%
      table_decouverte%(table%,decouverte%,2)=tresor%
      INC decouverte%
    LOOP
    EXIT IF info%=-1
    INC table%
    decouverte%=1
  LOOP
RETURN
'
'
' ****************************
' *  TIRAGE D'UNE DECOUVERTE  *
' ****************************
' ****************************************************
' *Cette routine détermine aléatoirement un numéro de*
' * trésor à partir de la table de découverte table%.*
' ****************************************************
FUNCTION tirage_decouverte(table%)
  LOCAL tirage%,decouverte%
  LOCAL prob%,tresor%
  LOCAL total_prob%
  '
  tirage%=RANDOM(100)+1
  tresor%=-1
  total_prob%=0
  FOR decouverte%=1 TO 30
    prob%=table_decouverte%(table%,decouverte%,1)
    IF prob%<>-1
      ADD total_prob%,prob%
      IF (tirage%<=total_prob%)
        tresor%=table_decouverte%(table%,decouverte%,2)
        decouverte%=30  ! Sortie de la boucle
      ENDIF
    ENDIF
  NEXT decouverte%
  '
  IF tresor%=-1
    PRINT "ERREUR DE FORMAT DANS LES DECOUVERTES"
    PRINT "NUMERO TABLE ";table%
  ENDIF
  RETURN tresor%
ENDFUNC
'
' ____________________________________________
' ********************************************
' *                                          *
' *  ROUTINES D'AFFICHAGE DE LA CARTE VISIBLE *
' *                                          *
' ********************************************
' *********************
' *  AFFICHAGE DE LA  *
' *   CARTE VISIBLE   *
' *********************
PROCEDURE aff_carte
  LOCAL l%,c%
  LOCAL ligne%
  LOCAL colonne%
  LOCAL num%
  LOCAL posx%
  LOCAL posy%
  '
  posy%=ycarte%
  ligne%=ligne_carte%
  FOR l%=1 TO 11
    posx%=xcarte%
    colonne%=colonne_carte%
    FOR c%=1 TO 11
      num%=carte|(colonne%,ligne%)
      PUT posx%,posy%,element$(num%)
      INC colonne%
      ADD posx%,16
    NEXT c%
    INC ligne%
    ADD posy%,16
  NEXT l%
  PUT xcarte%+80,ycarte%+80,perso$,1 ! Aff image perso
RETURN
'
' ************************************
' *  GESTION PERMUTATION DES ECRANS  *
' ************************************
PROCEDURE preparation_ecr_travail
  IF etat%=1
    ecr_visible%=adr1%
    ecr_travail%=adr2%
  ELSE
    ecr_visible%=adr2%
    ecr_travail%=adr1%
  ENDIF
  etat%=-etat%
  @set_ecr_log(ecr_travail%)
RETURN
'
' ******************************
' *  VISUALISATION DE LA CARTE *
' ******************************
PROCEDURE visualisation_ecr_travail
  @set_ecr_phys(ecr_travail%)
  VSYNC
RETURN
'
' ******************************
' *  DEPLACEMENT VERS LE HAUT  *
' ******************************
PROCEDURE aller_haut
  LOCAL ecr1%,x1%,y1%
```

```
  LOCAL ecr2%,x2%,y2%
  LOCAL px%,py%
  LOCAL c%,colonne%
  LOCAL n%
  LOCAL old_element%
  '
  old_element%=carte|(colonne_carte%+5,ligne_carte%+5)
  DEC ligne_carte%
  @preparation_ecr_travail
  ecr1%=ecr_visible%
  x1%=xcarte%
  y1%=ycarte%
  ecr2%=ecr_travail%
  x2%=xcarte%
  y2%=ycarte%+16
  RC_COPY ecr1%,x1%,y1%,176,160 TO ecr2%,x2%,y2%
  PUT xcarte%+80,ycarte%+96,element$(old_element%)
  colonne%=colonne_carte%
  px%=xcarte%
  py%=ycarte%
  FOR c%=1 TO 11
    n%=carte|(colonne%,ligne_carte%)
    PUT px%,py%,element$(n%)
    INC colonne%
    ADD px%,16
  NEXT c%
  PUT xcarte%+80,ycarte%+80,perso$,1 ! Aff image perso
  @copie_fenetre
  @visualisation_ecr_travail
RETURN
'
' ****************************
' *  DEPLACEMENT VERS LE BAS  *
' ****************************
PROCEDURE aller_bas
  LOCAL ecr1%,x1%,y1%
  LOCAL ecr2%,x2%,y2%
  LOCAL px%,py%
  LOCAL c%,colonne%
  LOCAL n%
  LOCAL old_element%
  '
  old_element%=carte|(colonne_carte%+5,ligne_carte%+5)
  INC ligne_carte%
  @preparation_ecr_travail
  ecr1%=ecr_visible%
  x1%=xcarte%
  y1%=ycarte%+16
  ecr2%=ecr_travail%
  x2%=xcarte%
  y2%=ycarte%
  RC_COPY ecr1%,x1%,y1%,176,160 TO ecr2%,x2%,y2%
  PUT xcarte%+80,ycarte%+64,element$(old_element%)
  colonne%=colonne_carte%
  px%=xcarte%
  py%=ycarte%+160
  FOR c%=1 TO 11
    n%=carte|(colonne%,ligne_carte%+10)
    PUT px%,py%,element$(n%)
    INC colonne%
    ADD px%,16
  NEXT c%
  PUT xcarte%+80,ycarte%+80,perso$,1 ! Aff image perso
  @copie_fenetre
  @visualisation_ecr_travail
RETURN
'
' ********************************
' *  DEPLACEMENT VERS LA GAUCHE  *
' ********************************
PROCEDURE aller_gauche
  LOCAL ecr1%,x1%,y1%
  LOCAL ecr2%,x2%,y2%
  LOCAL px%,py%
  LOCAL l%,ligne%
  LOCAL n%
  LOCAL old_element%
  '
  old_element%=carte|(colonne_carte%+5,ligne_carte%+5)
  DEC colonne_carte%
  @preparation_ecr_travail
  ecr1%=ecr_visible%
  x1%=xcarte%
  y1%=ycarte%
  ecr2%=ecr_travail%
  x2%=xcarte%+16
  y2%=ycarte%
  RC_COPY ecr1%,x1%,y1%,160,176 TO ecr2%,x2%,y2%
  PUT xcarte%+96,ycarte%+80,element$(old_element%)
  ligne%=ligne_carte%
  px%=xcarte%
  py%=ycarte%
  FOR l%=1 TO 11
    n%=carte|(colonne_carte%,ligne%)
    PUT px%,py%,element$(n%)
    INC ligne%
    ADD py%,16
  NEXT l%
  PUT xcarte%+80,ycarte%+80,perso$,1 ! Aff image perso
  @copie_fenetre
  @visualisation_ecr_travail
RETURN
'
' *********************************
' *  DEPLACEMENT VERS LA DROITE  *
' *********************************
PROCEDURE aller_droite
  LOCAL ecr1%,x1%,y1%
  LOCAL ecr2%,x2%,y2%
  LOCAL px%,py%
  LOCAL l%,ligne%
  LOCAL n%
  LOCAL old_element%
  '
```

```
  old_element%=carte|(colonne_carte%+5,ligne_carte%+5)
  INC colonne_carte%
  @preparation_ecr_travail
  ecr1%=ecr_visible%
  x1%=xcarte%+16
  y1%=ycarte%
  ecr2%=ecr_travail%
  x2%=xcarte%
  y2%=ycarte%
  RC_COPY ecr1%,x1%,y1%,160,176 TO ecr2%,x2%,y2%
  PUT xcarte%+64,ycarte%+80,element$(old_element%)
  ligne%=ligne_carte%
  px%=xcarte%+160
  py%=ycarte%
  FOR l%=1 TO 11
    n%=carte|(colonne_carte%+10,ligne%)
    PUT px%,py%,element$(n%)
    INC ligne%
    ADD py%,16
  NEXT l%
  PUT xcarte%+80,ycarte%+80,perso$,1 ! Aff image perso
  @copie_fenetre
  @visualisation_ecr_travail
RETURN
'
'
' ************************************************
' *                                              *
' *  FONCTIONS DE GESTION DES EVENEMENTS DU JEU  *
' *                                              *
' ************************************************
' ****************************************
' *  LE JOUEUR SE DEPLACE SUR LA PLAINE  *
' ****************************************
PROCEDURE rencontre_plaine
  LOCAL creature%
  LOCAL message$
  '
  IF @tst_probabilite(10)
    creature%=@tirage_rencontre(1)
    message$="RENCONTRE : "+creature$(creature%)
    @aff_fenetre(message$)
  ENDIF
RETURN
'
' ****************************************
' *  LE JOUEUR SE DEPLACE SUR LA FORET  *
' ****************************************
PROCEDURE rencontre_foret
  LOCAL creature%
  LOCAL message$
  '
  IF @tst_probabilite(15)
    creature%=@tirage_rencontre(2)
    message$="RENCONTRE : "+creature$(creature%)
    @aff_fenetre(message$)
  ENDIF
RETURN
'
'
' ********************************
' *  LE JOUEUR EXPLORE DES RUINES  *
' ********************************
' ****************************************
' *  Il peut rencontrer des créatures,  *
' *  ou trouver divers objets anciens.  *
' ****************************************
PROCEDURE rencontre_ruines
  LOCAL creature%
  LOCAL decouverte%
  LOCAL m$
  '
  IF @tst_probabilite(10)
    creature%=@tirage_rencontre(3)
    m$="RENCONTRE : "+creature$(creature%)
    @aff_fenetre(m$)
  ENDIF
  '
  IF @tst_probabilite(5)
    decouverte%=@tirage_decouverte(1)
    m$="DECOUVERTE :"+decouverte$(decouverte%)
    @aff_fenetre(m$)
  ENDIF
RETURN
'
'
' ************************************
' *    VISUALISATION DU DEPLACEMENT  *
' *  SUR LA CARTE AFFICHEE A L'ECRAN *
' ************************************
PROCEDURE exec_deplacement(direction%)
  SELECT direction%
  CASE 1
    @aller_haut
  CASE 2
    @aller_bas
  CASE 3
    @aller_droite
  CASE 4
    @aller_gauche
  ENDSELECT
RETURN
'
' ****************************
' *  ROUTINE DE DEPLACEMENT  *
' ****************************
PROCEDURE gestion_deplacement(direction%)
  LOCAL c%,l%,n%
  '
  SELECT direction%
  CASE 1
    c%=colonne_carte%+5
    l%=ligne_carte%+4
  CASE 2
```

```
    c%=colonne_carte%+5
    l%=ligne_carte%+6
  CASE 3
    c%=colonne_carte%+6
    l%=ligne_carte%+5
  CASE 4
    c%=colonne_carte%+4
    l%=ligne_carte%+5
  ENDSELECT
  '
  n%=def_ev%(carte|(c%,l%))
  '
  ' ********************************************
  ' *  EXECUTION DES EVENEMENTS DE DEPLACEMENT  *
  ' ********************************************
  SELECT n%
  CASE 1
    @exec_deplacement(direction%)
  CASE 2
    @exec_deplacement(direction%)
    @rencontre_plaine
  CASE 3
    @exec_deplacement(direction%)
    @rencontre_foret
  CASE 4
    @exec_deplacement(direction%)
    @rencontre_ruines
  ENDSELECT
RETURN
'
' ****************************
' *  EXECUTION DES FONCTIONS  *
' ****************************
PROCEDURE exec_fonction(f%)
  SELECT f%
  CASE 1,2,3,4
    @gestion_deplacement(f%)
  CASE 5
    @quitter_programme
  ENDSELECT
RETURN
'
' *******************************
' *  ATTENTE EVENEMENTS CLAVIER  *
' *******************************
' 1 : aller haut
' 2 : Aller bas
' 3 : Aller droite
' 4 : Aller gauche
' 5 : Quitter le programme
FUNCTION att_ev
  LOCAL c$,num_ev%
  '
  num_ev%=0
  @vider_clavier
  DO
    c$=INKEY$
    IF c$<>""
      IF c$=CHR$(27)
        num_ev%=5
      ENDIF
      IF LEN(c$)=2
        t%=ASC(MID$(c$,2,1))
        IF t%=72
          num_ev%=1
        ENDIF
        IF t%=80
          num_ev%=2
        ENDIF
        IF t%=77
          num_ev%=3
        ENDIF
        IF t%=75
          num_ev%=4
        ENDIF
      ENDIF
    ENDIF
    EXIT IF num_ev%<>0
  LOOP
  RETURN num_ev%
ENDFUNC
'
' **********************************
' *  GESTION DES ORDRES DU JOUEUR  *
' **********************************
PROCEDURE gestion_programme
  LOCAL ev%
  '
  DO
    ev%=@att_ev
    exec_fonction(ev%)
    EXIT IF sortie%=1
  LOOP
RETURN
'
' ****************************************
' *  LECTURE DES NUMEROS D'EVENEMENTS    *
' *  ASSOCIES AUX ELEMENTS DE LA CARTE   *
' ****************************************
PROCEDURE init_evenements
  LOCAL num%,val%
  '
  ARRAYFILL def_ev%(),0
  RESTORE def_evenements
  DO
    READ num%
    EXIT IF num%=-1
    READ val%
    def_ev%(num%)=val%
  LOOP
RETURN
'
' ******************************
' *  DEFINITION DES CREATURES  *
```

```
' ******************************
PROCEDURE init_creatures
  LOCAL num%,nom$
  '
  n%=0
  RESTORE def_creatures
  DO
    READ num%
    EXIT IF num%=-1
    READ nom$
    creature$(num%)=nom$
  LOOP
RETURN
'
' ********************************
' *  DEFINITION DES DECOUVERTES  *
' ********************************
PROCEDURE init_decouvertes
  LOCAL num%,nom$
  '
  n%=0
  RESTORE def_decouvertes
  DO
    READ num%
    EXIT IF num%=-1
    READ nom$
    decouverte$(num%)=nom$
  LOOP
RETURN
'
' **************************
' *  PROCEDURE PRINCIPALE  *
' **************************
PROCEDURE main
  nb_elements%=0
  ligne_carte%=1
  colonne_carte%=1
  element_courant%=1
  '
  xcarte%=8
  ycarte%=12
  @init_creatures
  @charge_perso("PERSO.BIN")
  @charge_elements("ELEMENTS.BIN")
  @charger_carte("carte.car")
  @init_evenements
  @init_rencontres
  @init_decouvertes
  @init_tables_decouvertes
  '
  HIDEM
  GRAPHMODE 2
  DEFTEXT 0
  '
  @cls_ecran(1)
  @cadre(xcarte%-1,ycarte%-1,178,178,1,0)
  @init_fenetre
  @aff_carte
  BMOVE XBIOS(2),adr2%,32000
  '
  etat%=1
  gestion_programme
  '
  SHOWM
RETURN
'
' ____________________________________
' ************************************
' *                                  *
' *  DEFINITION DES DONNEES DU JEU   *
' *                                  *
' ************************************
' ******************************
' *  DEFINITION DES EVENEMENTS  *
' ******************************
' *  DATA element,evenement     *
' *  DATA -1 : fin evenements   *
' ******************************
' *  Codages evenements :       *
' *  0 : déplacement impossible *
' *  1 : déplacement possible   *
' *  2 : Appel fonction 2       *
' *      (déplacement plaine)   *
' *  3 : Appel fonction 3       *
' *      (déplacement forêt)    *
' *  4 : Appel fonction 4       *
' *      (déplacement ruines)   *
' ******************************
def_evenements:
DATA 1,0
DATA 2,0
DATA 3,2
DATA 4,2
DATA 5,0
DATA 6,0
DATA 7,0
DATA 8,0
DATA 9,4
DATA 10,0
DATA 11,3
DATA -1
'
' ******************************
' *  DEFINITION DES CREATURES  *
' ******************************
def_creatures:
DATA 1,"DES LOUPS"
DATA 2,"DES CHIENS SAUVAGES"
DATA 3,"UN OURS"
DATA 4,"UN RENARD"
DATA 5,"SERPENT"
'
DATA 20,"DES VOLEURS"
DATA 21,"DES PAYSANS"
```

```
DATA 22,"UN COLPORTEUR"
DATA 23,"DES BUCHERONS"
DATA 24,"UNE CARAVANE"
DATA 25,"DES PILLEURS DE TOMBE"
'
DATA 50,"DES ORCS"
DATA 51,"DES GOBELINS"
DATA 52,"UN ELFE NOIR"
DATA 53,"UN NECROMAN"
'
DATA 70,"UNE ARAIGNEE GEANTE"
DATA 71,"UN SERPENT GEANT"
'
DATA 100,"UN DEMON"
DATA 101,"UN FANTOME"
DATA 102,"UN ESPRIT DES BOIS"
DATA 103,"UN ESPRIT DE L'EAU"
DATA 104,"UN ESPRIT DE TERRE"
DATA 105,"UN ESPRIT DE FEU"
DATA 106,"UN ESPRIT DE MORT"
DATA 107,"UN DRAGON NOIR"
DATA 108,"UN DRAGON ROUGE"
DATA 109,"UN DRAGON DES EAUX"
DATA -1
'
' ******************************************
' *   DEFINITION DES TABLES DE RENCONTRE   *
' ******************************************
' *  DATA créature, probabilité rencontre  *
' *  DATA -2 : passage table suivante      *
' *  DATA -1 : fin des tables              *
' *  REMARQUE : Le total des probabilités  *
' *  doit faire absolument 100%.           *
' ******************************************
def_rencontres:
' **********************
' *  TABLE 1 - PLAINE  *
' **********************
' *  Paysans    : 50%  *
' *  Colporteur : 10%  *
' *  Voleurs    : 20%  *
' *  Chiens     : 10%  *
' *  Orcs       : 10%  *
' **********************
DATA 21,50
DATA 22,10
DATA 20,20
DATA 2,10
DATA 50,10
DATA -2
' **********************
' *  TABLE 2 - FORET   *
' **********************
' *  Loups      : 30%  *
' *  Ours       : 15%  *
' *  Bucherons  : 10%  *
' *  Orcs       : 25%  *
' *  Gobelins   : 10%  *
' *  Esprit            *
' *  des bois   : 10%  *
' **********************
DATA 1,30
DATA 3,15
DATA 23,10
DATA 50,25
DATA 51,10
DATA 102,10
DATA -2
'
' ******************************
' *     TABLE 3 - RUINES       *
' ******************************
' *  Orcs               : 30%  *
' *  Gobelins           : 20%  *
' *  Serpent            : 20%  *
' *  Araignée géante    : 10%  *
' *  Pilleurs de tombes : 10%  *
' *  Fantome            : 05%  *
' *  Démon              : 05%  *
' ******************************
DATA 50,30
DATA 51,20
DATA 5,20
DATA 70,10
DATA 25,10
DATA 101,5
DATA 100,5
DATA -1
'
' *********************************
' *  DEFINITION DES DECOUVERTES   *
' *********************************
def_decouvertes:
DATA 1,"UNE PIECE D'OR"
DATA 2,"UNE PIECE D'ARGENT"
DATA 3,"UNE PIECE DE CUIVRE"
DATA 4,"DES PIECES D'OR"
DATA 5,"DES PIECES D'ARGENT"
DATA 6,"DES PIECES DE CUIVRE"
'
DATA 20,"UNE PIERRE PRECIEUSE"
DATA 21,"DES PIERRES PRECIEUSES"
'
DATA 50,"UNE VIEILLE EPEE"
DATA 51,"UN COUTEAU"
'
DATA 90,"UNE CHAUSSURE"
DATA 91,"UNE POUPEE DE CHIFFONS"
'
DATA 100,"UNE ARME MAGIQUE"
DATA -1
'
' ********************************************
' *  DEFINITION TABLE DE TIRAGE DES TRESORS  *
```

```
' ********************************************
tables_decouvertes:
' **********************
' *  TABLE 1 - RUINES  *
' **********************
' ***********************************
' *  Une piéce de cuivre     : 30%  *
' *  Des pièces de cuivre    : 15%  *
' *  Une piéce d'argent      : 15%  *
' *  Des pièces d'argent     : 07%  *
' *  Une pièce d'or          : 05%  *
' *  Des pièces d'or         : 03%  *
' *  Une pierre précieuse    : 03%  *
' *  Des pierres précieuses  : 02%  *
' *  Une épée                : 06%  *
' *  Un couteau              : 05%  *
' *  Une chaussure           : 04%  *
' *  Une poupée              : 04%  *
' *  Une arme magique        : 01%  *
' ***********************************
DATA 3,30
DATA 6,15
DATA 2,15
DATA 5,07
DATA 1,05
DATA 4,03
DATA 20,03
DATA 21,02
DATA 50,06
DATA 51,05
DATA 90,04
DATA 91,04
DATA 100,01
DATA -1
```


TELECHARGEZ DES CENTAINES ET DES CENTAINES DE LOGICIELS SUR LE 3615 ATARI

DE L'INITIATION A LA CRÉATION MUSICALE

Vous avez un 1040 STE

Vous êtes un fan de musique...

Equipez-vous à prix réduit

Le clavier BONTEMPI AZ 7500

Dernier-né de la gamme BONTEMPI, il est de loin le plus performant. Il saura vous satisfaire grâce à ses multiples qualités: Clavier MIDI «splitable» • 61 grandes touches • 32 sons «PCM Digital» • 24 rythmes PCM technologie numérique • 7 voies d'accompagnement, polyphonique (12 notes) • 4 effets sonores intégrés. Il s'harmonise idéalement avec le 1040 STE en exploitant toutes ses performances sur le plan musical. Le BONTEMPI AZ 7500 est l'outil qui vous permet de mettre en application vos progrès musicaux.

+

Le logiciel BIG BOSS Piano

Dernier-né de la société RYTHM'N SOFT, BIG BOSS Piano, est le «nec plus ultra» des logiciels musicaux: Cours de piano classique et moderne • clavier guide • défilement des partitions en temps réel • impression des partitions • quantisation • c'est aussi un séquenceur de 25 pistes permettant le chargement simultané de 10 morceaux. Le logiciel BIG BOSS Piano est une méthode éducative simple (assistance permanente) et rapide (10 leçons progressives) pour que le solfège ne soit plus une contrainte insurmontable. Laissez libre cours à votre inspiration et réveillez le musicien qui sommeille en vous.

Offre spéciale clavier + logiciel

1990 F TTC seulement
+ 110 F de frais d'expédition.
Livraison sous huitaine

Dans la limite des stocks disponibles.

Pour commander, remplissez lisiblement ce bon (ou une copie) et envoyez-le accompagné du règlement à :

ARTIPRESSE - 79, avenue Louis Roche - 92230 Gennevilliers

☐ Je commande le clavier BONTEMPI AZ 7500 + le logiciel BIG BOSS Piano au prix exeptionnel de 1 990 F + frais d'expédition. Ci-joint un chèque de 2 100 F en règlement.

Nom

Prénom

Adresse

Code postal

Ville

PROGRAMMATION D'UN JEU DE ROLE (IV)

Moi vois, moi tue

Nous vous présentons maintenant un programme permettant de simuler le déroulement d'un combat entre un personnage et une créature quelconque.

Combat, échange de coups

Le résultat ou l'échec d'une attaque doit être calculé en fonction des caractéristiques des adversaires. Pour écrire un programme performant, il faut analyser tout ce qui se passe pendant un combat et définir les caractéristiques des combattants nécessaires à une bonne simulation.
Cet article vous propose deux systèmes de combat. Le premier est un système simple, fourni à titre d'exemple. Le second est un mécanisme de combat plus complexe et plus réaliste. Il est accompagné d'une bibliothèque d'objets que vous pourrez utiliser dans vos propres programmes (armes, protections et créatures).

Système de combat de base

Le combat s'arrête lorsque l'un des deux adversaires est mort. Les combattants sont définis par une capacité de combat, une capacité de coups et un certain nombre de points de vie.
La capacité de combat est le pourcentage de chance que possède une créature de toucher son adversaire. Un combattant de bas niveau peut avoir un pourcentage faible (entre 5% et 15%) alors qu'un combattant de haut niveau ou une créature très puissante peuvent avoir un pourcentage très élevé (entre 70% et 90%). Pour savoir si une créature touche son ennemi, il faut tirer un nombre aléatoire compris entre 1 et 100 (avec l'instruction RANDOM). Si ce chiffre est inférieur ou égal à la capacité de combat, l'attaque est réussie.
La capacité de coups est le nombre de points de dégâts qu'une créature peut causer à une autre. Un éléphant fait nettement plus de dommages qu'un écureuil.
Le nombre de points de vie représente la vigueur d'une créature et sa capacité à encaisser des coups. Les points de vie diminuent lorsque la créature reçoit des points de dommages à la suite d'une attaque. Si le nombre de points de vie tombe à 0, la créature est morte.

```
' vie_perso%        : points de vie Perso
' combat_perso%     : capacité de combat Perso
' coups_perso%      : capacité de coups Perso
' vie_creature%     : Points de vie Créature
' combat_creature%  : Capacité de combat Créature
' coups_creature%   : Capacité de coups Créature
'
DO
  ' *************************
  ' * Perso attaque Créature *
  ' *************************
  IF RANDOM(100)<=combat_perso%
    SUB vie_creature%,coups_perso%
  ENDIF
  EXIT IF vie_creature%<=0
  '
  ' *************************
  ' * Créature attaque Perso *
  ' *************************
  IF RANDOM(100)<=combat_creature%
    SUB vie_perso%,coups_creature%
  ENDIF
  EXIT IF vie_perso%<=0
LOOP
'
' ********************************
' *  Affichage résultat du combat  *
' ********************************
IF vie_perso%<=0
  PRINT "PERSO est mort"
ELSE
  PRINT "CREATURE est mort(e)"
ENDIF
```

Système de combat plus évolué

Le système de combat de base est simple à mettre en oeuvre, mais peu performant. Nous vous présentons maintenant un système de combat sophistiqué qui permet de réaliser des combats plus réalistes. Les créatures sont définies par les caractéristiques suivantes: Force, Points de vie, Capacité de combat, Nombre d'attaques par round de combat, Compétence de défense, Arme et Protection.

Les caractéristiques Points de vie et Capacité de combat ont la même signification que dans le système de combat de base.
Le Nombre d'attaques est le nombre de fois qu'un combattant peut attaquer dans le laps de temps d'un round de combat. Une créature faible ne peut attaquer qu'une fois alors qu'une créature très puissante peut attaquer de nombreuses fois par tour. Cela dépend de l'entraînement et de la vitesse de réaction d'un combattant, voire même de sa structure physique (une pieuvre avec plusieurs bras peut exécuter plusieurs attaques en même temps).
La Capacité de défense représente la possibilité d'éviter une attaque. Cela peut être dû à une grande agilité, à une petite taille ou à une structure physique particulière (une créature faite de brume qui devient solide uniquement au moment de frapper est très difficile à toucher).
Cela peut aussi être le résultat d'un entraînement spécifique : un combattant ayant appris à esquiver ou à parer une attaque peut avoir une bonne capacité de défense.
La Capacité de coups du système de base est remplacée par les caractéristiques Force et Arme. Le programme calcule les points de dommages en tenant compte du type d'arme utilisé et de la force de la créature.
La caractéristique Protection correspond à une éventuelle protection corporelle (vêtement solide, armure, écailles, peau épaisse, etc.) Cette protection diminue les dommages reçus en combat. Elle peut même les annuler totalement dans certains cas.

Déroulement d'un combat

Le combat est divisé en rounds de combat. Pendant un round de combat, les deux combattants frappent l'un après l'autre jusqu'à épuisement de leur nombre d'attaques. Lorsque l'un des adversaires est touché, le programme calcule le nombre de points de dégâts en tenant compte de la force et de l'arme de l'attaquant, ainsi que de la protection portée par la cible.
Le pourcentage de toucher une cible est calculé en prenant la capacité de combat de l'attaquant et en soustrayant la capacité de défense de la cible.
Les dégâts sont retirés aux points de vie du combattant touché. Le combat cesse lorsque l'un des deux adversaires est tué, c'est-à-dire lorsque son nombre de points de vie est égal ou inférieur à 0.

Conclusion d'un combat

A la fin d'un combat, le programme affiche le résultat. Si le personnage a gagné, il reçoit le trésor éventuellement possédé par la créature et des points d'expérience.

Les points d'expérience

Un personnage qui vit de nombreuses aventures acquiert de l'expérience et augmente ses compétences. Pour simuler cela, un personnage victorieux reçoit un certain nombre de points d'expérience. Lorsque son total de points dépasse une certaine limite, ses compétences augmentent. Le nombre de points reçu dépend de la créature. Vaincre un lapin rapporte peu d'expérience, alors que vaincre un dragon rapporte énormément de points, à condition que le personnage survive.

Programmation du système de combat

La première chose à faire avant d'écrire le programme est d'étudier la structure des objets stockés en mémoire et de décider de la méthode utilisée pour définir ces objets. Ces derniers sont: les armes, les moyens de protection, les trésors, les créatures et le personnage.
Il est très difficile de déterminer les valeurs numériques définissant un objet. La tâche est simplifiée si l'on considère qu'un homme moyen possède une base de 10 à toute ces caractéristiques (Force de 10, Intelligence de 10, Vitesse de réaction de 10, 10 Points de vie, etc.) Il faut ensuite calculer toutes les données du jeu en tenant compte de cette base. Par exemple, un chat est nettement moins fort qu'un homme, mais plus qu'un lapin. On donnera 1 de force au lapin et 3 de force au chat. Par contre, un lion est plus fort qu'un homme. On peut lui donner une force de 14.

Définition des armes

Les armes sont des objets qui servent à causer des dégâts physiques sur une cible quelconque (un être vivant le plus souvent). Les dommages causés dans un combat comportent un élément aléatoire. Une épée peut tout aussi bien provoquer une blessure mineure qu'une blessure grave en frappant un organe vital. Pour simuler cela, il faut faire la distinction entre les dommages constants et les dommages varaibles.
Une arme est définie par un nom, un dommage constant et un dommage variable. Par exemple, une épée cause 2 points de dommage constant et 3 points de dommage variable. Un coup d'épée peut donc causer entre 2 (2+1) et 5 (2+3) points de dommage. Cela donne une moyenne de 4 points de dommage. La valeur des points de dommage a été définie en tenant compte qu'un homme moyen possède 10 points de vie.
Les noms des armes sont stockés dans le tableau `nom_arme$(n%)`. Les dommages constants sont mémorisés dans le tableau `def_arme%(n%,1)` et les variables dans le tableau `def_arme%(n%,2)`.

```
DIM nom_arme$(100)
DIM def_arme%(100,2)
' def_arme%(n,1) = dégât constant arme n
' def_arme%(n,2) = dégât variable arme n
```

La définition des armes est faite dans des lignes de `DATA`. Elles sont définies sous la forme `numéro, "nom", dommage constant, dommage variable`. La donnée `numéro` est le numéro de référence de l'arme. Par exemple, le nom de l'arme 12 est stocké dans le tableau `nom_arme$(12)`, le dommage constant dans `def_arme%(12,1)` et le dommage variable dans `def_arme%(12,2)`.

```
' *************************
' *  DEFINITION DES ARMES  *
' *************************
'DATA numéro,"nom", dommage constant, dommage variable
armes_data:
DATA 1,"GRIFFES",2,2
DATA 2,"CROCS",2,3
DATA 3,"MORSURES EMPOISONNES",1,5
DATA 3,"TENTACULES",1,4
```

```
DATA 4,"SOUFFLE DE FEU",30,40
' --
DATA 10,"COUTEAU",0,2
DATA 11,"EPEE",2,3
DATA 12,"EPEE LONGUE",2,4
DATA 13,"MASSE",2,3
DATA 14,"HACHE",2,2
DATA 15,"HACHE DE GUERRE",2,4
DATA 16,"FLEAU",3,3
DATA 17,"LANCE",1,4
DATA 18,"CIMETERE",2,4
' --
DATA -1
```

Ces lignes contiennent quelques exemples d'armes classiques dans un environnement médiéval-fantastique. Les premières sont des armes naturelles utilisées par diverses créatures. Les suivantes sont des armes manufacturées utilisées par des êtres intelligents comme les humains et les créatures plus ou moins exotiques habitant les univers fantastiques (orques, elfes, nains, gobelins, etc.)
La procédure @init_armes lit les informations stockées dans les lignes de DATA et les accumule dans les tableaux nom_arme$() et def_arme%(). Elle doit être appelée en début de programme.

```
PROCEDURE init_armes
  LOCAL nom$
  LOCAL degat_constant%
  LOCAL degat_variable%
  LOCAL n%
  '
  RESTORE armes_data
  ARRAYFILL def_arme%(),0
  DO
    READ n%
    EXIT IF n%=-1
    READ nom$
    READ degat_constant%
    READ degat_variable%
    nom_arme$(n%)=nom$
    def_arme%(n%,1)=degat_constant%
    def_arme%(n%,2)=degat_variable%
  LOOP
RETURN
```

La procédure @dommage_arme(num_arme%) calcule le dommage causé par l'arme num_arme%. Les dommages variables sont calculés par la formule RANDOM(prob_degat%+1). Pourquoi cette formule? Parce que RANDOM(n) fournit un nombre aléatoire compris entre 0 et n-1. Pour obtenir un nombre aléatoire compris entre 0 et n, il faut utiliser la formule RANDOM(n+1).

```
' ***********************************
' *  CALCUL DOMMAGE ARME num_arme%  *
' ***********************************
FUNCTION dommage_arme(num_arme%)
  LOCAL degat_constant%
  LOCAL prob_degat%
  LOCAL degat_variable%
  '
  degat_constant%=def_arme%(num_arme%,1)
  prob_degat%=def_arme%(num_arme%,2)
  degat_variable%=RANDOM(prob_degat%+1)
  RETURN degat_constant%+degat_variable%
ENDFUNC
```

Les protections

Les protections sont des vêtements ou des armures portés par les créatures du jeu. Elles assurent une fonction défensive. En terme de jeu, elles diminuent les dommages reçus par leurs utilisateurs.
Une protection est définie par un nom et une valeur de protection. Pour simplifier la programmation, la valeur de protection est constante. Toutefois, vous pouvez modifier le programme pour obtenir une protection variable.

```
DIM nom_protection$(100)
DIM def_protection%(100)
' def_protection$(n%) = nom protection n%
' def_protection%(n%) = valeur protection n%
```

Les protections sont définies dans des lignes de DATA. La liste suivante comprend de nombreux exemples de protection. La valeur -1 est l'indicateur de fin de données.

```
' ********************************
' *  DEFINITION DES PROTECTIONS  *
' ********************************
protections_data:
DATA 1,"FOURRURE",1
DATA 2,"FOURRURE EPAISSE",2
DATA 3,"ECAILLES",2
DATA 4,"PLAQUES D'OS",4
DATA 5,"PEAU DE DRAGON",50
DATA 6,"PEAU DE DEMON",10
'
DATA 20,"HABITS DE MENDIANT",0
DATA 21,"HABITS DE COLPORTEUR",0
DATA 22,"VETEMENTS ORDINAIRE",0
DATA 23,"VETEMENTS DE CUIR",1
DATA 24,"ROBE DE PRETRE",0
DATA 25,"HABITS DE NOBLE",0
DATA 26,"VETEMENTS DE MARCHAND",0
DATA 27,"ROBE MAGIQUE",4
'
DATA 40,"ARMURE DE CUIR",2
DATA 41,"COTTE DE MAILLES",3
DATA 42,"ARMURE DE PLAQUES",4
DATA 43,"ARMURE DRACONIQUE",30
DATA 44,"ARMURE CHAOTIQUE",100
'
DATA -1
```

La procédure @init_protections charge les données stockées en DATA dans les tableaux nom_protection$() et def_protection%().

```
PROCEDURE init_protections
  LOCAL nom$
  LOCAL protect%
  LOCAL n%
  '
  RESTORE protections_data
  ARRAYFILL def_protection%(),0
  DO
    READ n%
    EXIT IF n%=-1
    READ nom$
    READ protect%
    nom_protection$(n%)=nom$
    def_protection%(n%)=protect%
  LOOP
RETURN
```

La fonction @valeur_protection(objet%) fournit le nombre de points de protection fournis par l'objet objet%.

```
' ************************************
' *  VALEUR DE PROTECTION D'UN OBJET  *
' ************************************
FUNCTION valeur_protection(objet%)
  RETURN def_protection%(objet%)
ENDFUNC
```

Au niveau technique, elle se contente de renvoyer la valeur de la variable def_protection%(objet%). Cela peut paraître stupide d'écrire une fonction pour une opération aussi simple, mais cela permet d'obtenir une interface entre le programme et les données indépendantes de la technique utilisée pour stocker les données. Si vous modifiez la structure de stockage des protections, il suffira de modifier la fonction @valeur_protection sans toucher au reste du programme pour que tout continue de fonctionner normalement.

Les trésors

Certaines créatures intelligentes peuvent posséder des objets précieux. Si le personnage les tue, il récupère ces objets. Dans un but de simplification, le programme ne gère que la description des trésors. Elles sont stockées dans le tableau nom_tresor$().

```
DIM nom_tresor$(100)
```

Les descriptions des trésors sont définies dans des lignes de DATA. Elles sont stockées sous la forme numéro trésor, "description". Le numéro -1 correspond à la fin de données.

```
' ****************************
' *  DEFINITIONS DES TRESORS  *
' ****************************
' DATA type trésor, "description"
' DATA -1 = fin des trésors
'
tresors_data:
DATA 1,"QUELQUES PIECES DE CUIVRES"
DATA 2,"QUELQUES PIECES D'OR"
DATA 3,"DES CENTAINES DE PIECES D'OR"
DATA 4,"UN TRESOR FABULEUX"
DATA -1
```

La procédure @init_tresors initialise le tableau nom_tresor$() avec la description des trésors stockées en DATA.

```
PROCEDURE init_tresors
  LOCAL n%,tresor$
  '
  RESTORE tresors_data
  DO
    READ num%
    EXIT IF num%=-1
    READ tresor$
    nom_tresor$(num%)=tresor$
  LOOP
RETURN
```

Il est impossible de créer une fonction @description_tresor(tresor%) renvoyant la description du trésor tresor% car le *GFA Basic* n'admet pas les fonctions renvoyant une variable alphanumérique.

```
FUNCTION description_tresor(tresor%)
  RETURN nom_tresor$(tresor%)
ENDFUNC
```

Cette fonction provoque l'apparition d'un message d'erreur RETURN sans PROCEDURE La raison pour laquelle le système affiche un pareil message d'erreur n'est pas évident à première vue. Etant donné que la syntaxe du *GFA Basic* ne comprend pas l'instruction RETURN suivie d'une variable alphanumérique, l'interpréteur pense que le RETURN est la fin d'une procédure. Et comme il ne trouve pas l'instruction PROCEDURE correspondant à RETURN (puisque c'est une FUNCTION), il affiche un message d'erreur. Pour écrire une routine renvoyant la description d'un trésor, il faut utiliser l'instruction VAR qui permet de modifier la valeur d'un paramètre d'une procédure.

```
PROCEDURE description_tresor(tresor%,VAR tresor$)
  tresor$=nom_tresor$(tresor%)
RETURN
```

Pour écrire la description du trésor numéro 2 dans la variable d$, il faut utiliser la formule @description_tresor(2,d$).

```
' ************************
' *  AFFICHAGE TRESOR 2  *
' ************************
@description_tresor(2,d$)
PRINT d$
```

Définition des créatures

Les créatures sont définies par un nom, une Force, un nombre de Points de vie, une Capacité de combat, un Nombre d'attaques, une

Capacité de défense, un numéro d'arme, un numéro de protection, un nombre de points d'expérience et un numéro de trésor. Le nom des créatures est stocké dans le tableau nom_creature$() et les caractéristiques dans le tableau def_creature%().

```
DIM nom_creature$(100)
DIM def_creature%(100,9)
'
' def_creature%(n%,1) = Force
' def_creature%(n%,2) = points de vie
' def_creature%(n%,3) = Capacité de Combat
' def_creature%(n%,4) = Nb attaques
' def_creature%(n%,5) = Capacité de défense
' def_creature%(n%,6) = Numéro arme
' def_creature%(n%,7) = Numéro Protection
' def_creature%(n%,8) = Points expérience
' def_creature%(n%,9) = Type de trésor
```

Les caractéristiques des créatures sont stockées dans des lignes de DATA. Chaque créature possède un numéro de référence. Les créatures 1 a 19 sont des animaux. Les créatures 20 a 49 sont des créatures intelligentes. Les créatures 50 a 100 sont des créatures surnaturelles ou extrêmement puissantes. Tous les numéros de référence ne sont pas attribués. Il reste de la place pour vos propres monstres. Remarque: si une créature ne possède pas de trésor, sa caractéristique trésor contient la valeur 0.

```
' ******************************
' *  DEFINITION DES CREATURES  *
' ******************************
' DATA Numéro, "Nom", Force, Points de vie
' DATA Combat (%), Nb attaques, Défense (%)
' DATA Arme, Protection
' DATA Points d'expériences, Trésor
' DATA -1 = fin des créatures
creatures_data:
'
' *************
' *  1: OURS  *
' *************
' FORCE:18;     VIE:30
' COMBAT:68%;   ATTAQUE:2;  DEFENSE:50%;
' ARME:Griffes; PROTECTION:Fourrure épaisse
' EXP:1000;     TRESOR:Aucun
' ----
DATA 1,"OURS",18,30
DATA 68,2,50
DATA 1,2
DATA 1000,0
'
' ******************
' *  2: LOUP GRIS  *
' ******************
' FORCE:13;   VIE:12
' COMBAT:80%; ATTAQUE:1;  DEFENSE:40%
' ARME:Crocs; PROTECTION:Fourrure épaisse
' EXP:200;    TRESOR:Aucun
' ----
DATA 2,"LOUP GRIS",13,12
DATA 80,1,40
DATA 2,2
DATA 200,0
'
' ***************
' *  3:SERPENT  *
' ***************
' FORCE:10;  VIE:16
' COMBAT:60%;  ATTAQUE:1;  DEFENSE:30
' ARME:Morsure; PROTECTION:Ecailles;
' EXP:400;  TRESOR;Aucun
DATA 3,"SERPENT",10,16
DATA 60,1,30
DATA 3,3
DATA 400,0
'
' ****************************
' *  20 : BARBARE DU DESERT  *
' ****************************
' FORCE:14;       POINTS DE VIE:13
' COMBAT:60%;     ATTAQUE:2; DEFENSE:35
' ARME:Cimeterre;  PROTECTION:Armure de cuir
' EXP:800; TRESOR:Moyen
' ----
DATA 20,"BARBARE DU DESERT",14,13
DATA 60,2,35
DATA 18,40
DATA 800,2
'
' ***************
' *  21: ORQUE  *
' ***************
' FORCE:13;   POINTS DE VIE:14
' COMBAT:62;  ATTAQUE:1; DEFENSE:50%
' ARME:Masse; PROTECTION:Armure de cuir
' EXP:150;    TRESOR:Faible
' ----
DATA 21,"ORQUE",13,14
DATA 62,1,50
DATA 13,40
DATA 150,1
'
' *******************
' *  22: ELFE NOIR  *
' *******************
' FORCE:12;  POINTS DE VIE:16
' COMBAT:90%  ATTAQUE:3;  DEFENSE:50
' ARME:Epée longue;  PROTECTION:Armure de plaques
' EXP:2000;  TRESOR:Moyen
DATA 22,"ELFE NOIR",12,16
DATA 90,3,50
DATA 12,42
DATA 2000,2
'
' ***************
```

```
' *  50: DEMON  *
' ***************
' FORCE:20;  POINTS DE VIE:50
' COMBAT:90; ATTAQUE:5;  DEFENSE:80
' ARME:Griffes; PROTECTION:Peau de démon
' EXP:10000;  TRESOR:Aucun
DATA 50,"DEMON",20,50
DATA 90,5,80
DATA 1,6
DATA 10000,0
'
' ****************
' *  51: DRAGON  *
' ****************
' FORCE:30     POINTS DE VIE:500
' COMBAT:100   ATTAQUE:10  DEFENSE:90
' ARME:Souffle de feu;
' PROTECTION: Peau de Dragon
' EXP:90000;  TRESOR:Fabuleux
' ----
DATA 51,"DRAGON",30,500
DATA 100,10,90
DATA 4,5
DATA 90000,4
'
DATA -1
```

La procédure @init_creatures remplit le tableau def_creature%() avec les créatures définies dans les lignes de DATA.

```
PROCEDURE init_creatures
  LOCAL n%,nom$,force%,points_vie%
  LOCAL combat%,nb_attaques%,defense%
  LOCAL arme%,protection%
  LOCAL experience%,tresor%
  '
  RESTORE creatures_data
  DO
    READ n%
    EXIT IF n%=-1
    READ nom$
    READ force%
    READ points_vie%
    READ combat%
    READ nb_attaques%
    READ defense%
    READ arme%
    READ protection%
    READ experience%
    READ tresor%
    '
    nom_creature$(n%)=nom$
    def_creature%(n%,1)=force%
    def_creature%(n%,2)=points_vie%
    def_creature%(n%,3)=combat%
    def_creature%(n%,4)=nb_attaques%
    def_creature%(n%,5)=defense%
    def_creature%(n%,6)=arme%
    def_creature%(n%,7)=protection%
    def_creature%(n%,8)=experience%
    def_creature%(n%,9)=tresor%
  LOOP
RETURN
```

Les fonctions suivantes permettent au programme d'obtenir des informations sur une créature sans s'occuper de la structure de stockage des données. Elles simplifient la programmation et augmentent la lisibilité du programme source.

```
FUNCTION force_creature(creature%)
  RETURN def_creature%(creature%,1)
ENDFUNC

FUNCTION vie_creature(creature%)
  RETURN def_creature%(creature%,2)
ENDFUNC

FUNCTION combat_creature(creature%)
  RETURN def_creature%(creature%,3)
ENDFUNC

FUNCTION nb_attaques_creature(creature%)
  RETURN def_creature%(creature%,4)
ENDFUNC

FUNCTION defense_creature(creature%)
  RETURN def_creature%(creature%,5)
ENDFUNC

FUNCTION arme_creature(creature%)
  RETURN def_creature%(creature%,6)
ENDFUNC

FUNCTION protection_creature(creature%)
  RETURN def_creature%(creature%,7)
ENDFUNC

FUNCTION experience_creature(creature%)
  RETURN def_creature%(creature%,8)
ENDFUNC

FUNCTION tresor_creature(creature%)
  RETURN def_creature%(creature%,9)
ENDFUNC
```

Définition du personnage

Le personnage est géré d'une manière légèrement différente de celle des créatures. Il possède une donnée supplémentaire: le nombre de points de vie temporaire. Ces points de vie diminuent lorsque le personnage reçoit un coup dans un combat. Si le nombre de points de vie temporaire tombe à 0, le personnage est mort.

Les créatures ne possèdent pas de points de vie temporaires car leurs caractéristiques sont informatives. Lorsque le programme doit gérer un combat entre le personnage et une créature, il recopie dans une

variable spéciale le nombre de points de vie de la créature et décrémente cette variable lorsque la créature reçoit un coup. Ce système évite de modifier la valeur des caractéristiques stockées en mémoire. Si le personnage rencontre un loup peu de temps après avoir tué un premier loup, les caractéristiques du second doivent être identiques à celles du premier (et tout particulièrement le nombre de points de vie).

Le nom du personnage est stocké dans la variable nom_perso$. Les caractéristiques sont stockées dans le tableau def_perso%(). La caractéristique Expérience correspond au nombre de points d'expérience accumulé par le personnage lors de ses affrontements. Elle vaut 0 au début du programme.

```
nom_perso$=""
DIM def_perso%(9)
' def_perso%(1) = Force
' def_perso%(2) = Points de vie max
' def_perso%(3) = Points de vie temporaire
' def_perso%(4) = Compétence de combat (%)
' def_perso%(5) = Nb attaques
' def_perso%(6) = Compétence de défense (%)
' def_perso%(7) = Expérience
' def_perso%(8) = Arme
' def_perso%(9) = Protection
```

Les statistiques du personnage sont définies dans des lignes de DATA. Etant donné le petit nombre de caractéristiques, elles auraient pu être écrites directement dans les variables concernées, mais c'est une méthode de programmation à éviter.

```
' DATA Nom, Force, Points de vie
' DATA Combat (%), Nb attaques, Défense (%)
' DATA Arme, Protection
' ----
' nom:KURT BRISE-CRANE
' FORCE:13;  VIE:15;
' COMBAT:70%;  ATTAQUE:1;  DEFENSE:60%
' ARME:EPEE LONGUE; Protection: COTTE DE MAILLES
' ----
perso_data:
DATA "KURT BRISE-CRANE",13,15
DATA 70,1,60
DATA 12,41
```

La procédure @init_perso initialise la variable nom_perso$ et le tableau def_perso%() avec les informations définissant le personnage.

```
PROCEDURE init_perso
  LOCAL nom$,force%,points_vie%
  LOCAL combat%,nb_attaques%,defense%
  LOCAL arme%,protection%
  '
  RESTORE perso_data
  '
  READ nom$
  READ force%
  READ points_vie%
  READ combat%
  READ nb_attaques%
  READ defense%
  READ arme%
  READ protection%
  '
  nom_perso$=nom$
  def_perso%(1)=force%
  def_perso%(2)=points_vie%
  def_perso%(3)=points_vie%
  def_perso%(4)=combat%
  def_perso%(5)=nb_attaques%
  def_perso%(6)=defense%
  def_perso%(7)=0
  def_perso%(8)=arme%
  def_perso%(9)=protection%
RETURN
```

Les fonctions suivantes permettent au programme d'accéder facilement aux données définissant le personnage. De la même manière que les fonctions d'informations sur les créatures, elles permettent d'accéder aux informations du personnage sans rien connaître de leurs structures de stockage.

```
FUNCTION force_perso
  RETURN def_perso%(1)
ENDFUNC

FUNCTION vie_max_perso
  RETURN def_perso%(2)
ENDFUNC

FUNCTION vie_perso
  RETURN def_perso%(3)
ENDFUNC

FUNCTION combat_perso
  RETURN def_perso%(4)
ENDFUNC

FUNCTION nb_attaques_perso
  RETURN def_perso%(5)
ENDFUNC

FUNCTION defense_perso
  RETURN def_perso%(6)
ENDFUNC

FUNCTION experience_perso
  RETURN def_perso%(7)
ENDFUNC

FUNCTION arme_perso
  RETURN def_perso%(8)
ENDFUNC

FUNCTION protection_perso
```

```
  RETURN def_perso%(9)
ENDFUNC
```

Bonus aux dommages

Un homme fort fait plus de dégâts avec une épée qu'un frêle garçon utilisant la même arme. Les dégâts supplémentaires viennent de la différence de force entre l'homme et le garçon. Pour simuler l'effet de la force sur les dommages, le programme calcule le dommage d'une attaque en additionnant le dommage de l'arme et une valeur appelé «bonus de force». Le bonus de force dépend de la force de l'attaquant. Plus un personnage est fort, plus son bonus de dommage est important. De la même manière, plus un personnage est faible, moins son bonus de Force est élevé. Une créature vraiment faible peut même avoir un bonus de dommage négatif. La manière la plus simple de gérer les bonus de force est de les stocker dans le tableau bonus_dommage%(). Pour connaître le bonus de force de la force f%, il suffit de lire la valeur de bonus_dommage%(f%).

```
DIM bonus_dommage%(30)
```

La valeur des bonus est stockée dans des lignes de DATA. Les bonus ont été établis en tenant compte qu'un homme avec une force de 10 n'a aucun bonus. Ils sont définis sous la forme «Force/Bonus». La valeur -1 est l'indicateur de fin de données. Si vous voulez étendre la table de bonus au-delà de 30, il faut augmenter la taille du tableau bonus_dommage%().

```
bonus_dommages_data:
DATA 1,-5,   2,-4,   3,-3,   4,-2,   5,-1
DATA 6,-1,   7,0,    8,0,    9,0,    10,0
DATA 11,0,   12,1,   13,1,   14,2,   15,3
DATA 16,4,   17,5,   18,6,   19,7,   20,8
DATA 21,9,   22,10,  23,11,  24,12,  25,13
DATA 26,14,  27,15,  28,16,  29,17,  30,19
DATA -1
```

@init_bonus charge les bonus dans le tableau bonus_dommage%().

```
PROCEDURE init_bonus
  LOCAL n%,bonus%
  '
  ARRAYFILL bonus_dommage%(),0
  RESTORE bonus_dommages_data
  DO
    READ n%
    EXIT IF n%=-1
    READ bonus%
    bonus_dommage%(n%)=bonus%
  LOOP
RETURN
```

La fonction @bonus_force(force%) fournit le bonus de force correspondant à la force force%.

```
' ******************************
' *  VALEUR DU BONUS DE FORCE  *
' ******************************
FUNCTION bonus_force(force%)
  RETURN bonus_dommage%(force%)
ENDFUNC
```

Initialisation des données du programme

La procédure @init_datas initialise les tableaux de données avec la définition des différents objets du jeu (armes, protections, trésors, etc.) Elle doit impérativement être appelée en début de programme.

```
PROCEDURE init_datas
  @init_armes
  @init_protections
  @init_tresors
  @init_bonus
  @init_perso
  @init_creatures
RETURN
```

Exécution d'un combat

Pour exécuter un combat, le programme a besoin de connaître le type de la créature affrontant le personnage et son nombre de points de vie. La procédure @exec_combat(creature%) simule un combat entre le personnage et une créature de type creature%. Le type de créature est stocké dans la variable globale typ_ennemi% et son nombre de points de vie dans la variable globale vie_ennemi%.
Dés que l'un des combattants est mort, la routine @exec_combat affiche le résultat du combat. Si le personnage est vivant, elle affiche le nombre de points de vie restant, le nombre de points d'expérience gagnés et le type de trésor éventuellement possédé par la créature.

```
PROCEDURE exec_combat(creature%)
  LOCAL m$
  LOCAL tresor%
  LOCAL tresor$
  LOCAL exp%
  '
  typ_ennemi%=creature%
  vie_ennemi%=@vie_creature(creature%)
  @combat
  '
  IF @mort_ennemi=1
    m$=nom_creature$(typ_ennemi%)
    m$=m$+" est mort"
    PRINT
    PRINT m$
    tresor%=@tresor_creature(typ_ennemi%)
    exp%=@experience_creature(typ_ennemi%)
    IF tresor%<>0
      @description_tresor(tresor%,tresor$)
      PRINT "TRESOR : ";tresor$
    ENDIF
    PRINT "EXPERIENCE : ";exp%
    '
    PRINT "Points de vie perso : ";
```

```
    PRINT @vie_perso;"/";@vie_max_perso
    PRINT
  ENDIF
  '
  IF @mort_perso=1
    PRINT
    PRINT nom_perso$+" est mort"
  ENDIF
RETURN
```

La fonction @mort_ennemi teste si l'adversaire du personnage est mort en regardant la valeur contenue dans la variable vie_ennemi%. Elle renvoie 1 si l'adversaire est mort et 0 dans le cas contraire.

```
' ******************************************
' *  TEST SI L'ADVERSAIRE DU PERSO EST MORT  *
' ******************************************
FUNCTION mort_ennemi
  LOCAL result%
  '
  IF vie_ennemi%<=0
    result%=1
  ELSE
    result%=0
  ENDIF
  RETURN result%
ENDFUNC
```

La fonction @mort_perso teste si le personnage est mort. Elle utilise la routine @vie_perso pour connaître le nombre de points de vie restant du personnage. Elle retourne 1 si le personnage est mort et 0 dans le cas contraire.

```
' ************************************
' *  TEST SI LE PERSONNAGE EST MORT  *
' ************************************
FUNCTION mort_perso
  LOCAL result%
  '
  IF @vie_perso<=0
    result%=1
  ELSE
    result%=0
  ENDIF
  RETURN result%
ENDFUNC
```

Gestion du combat

La procédure @combat gère l'enchaînement des rounds de combat. La procédure @round_combat simule l'exécution d'un round. Le combat ne s'arrête que lorsque l'un des combattants est mort.

```
' ***********************
' *  GESTION DU COMBAT  *
' ***********************
PROCEDURE combat
  DO
    @round_combat
    EXIT IF @mort_perso=1
    EXIT IF @mort_ennemi=1
  LOOP
RETURN
```

Exécution d'un round de combat

La procédure @round_combat gère l'exécution d'un round de combat. Les adversaires frappent les uns après les autres jusqu'à épuisement de leur nombre d'attaques. Le combat cesse immédiatement si l'un des combattants meurt.

La routine lit le nombre d'attaque de chaque combattant, puis les exécute à la suite. La variable nb_att_perso% contient le nombre d'attaques du personnage. Le nombre d'attaques de son adversaire est stocké dans la variable nb_att_ennemi%. Le contenu de ces variables est décrémenté après chaque attaque.

```
' ************************************
' *  EXECUTION D'UN ROUND DE COMBAT  *
' ************************************
PROCEDURE round_combat
  LOCAL nb_att_perso%
  LOCAL nb_att_ennemi%
  '
  nb_att_perso%=@nb_attaques_perso
  nb_att_ennemi%=@nb_attaques_creature(typ_ennemi%)
  '
  REPEAT
    IF nb_att_perso%<>0
      @attaque_ennemi
      DEC nb_att_perso%
    ENDIF
    EXIT IF @mort_ennemi=1
    '
    IF nb_att_ennemi%<>0
      @attaque_perso
      DEC nb_att_ennemi%
    ENDIF
    EXIT IF @mort_perso=1
  UNTIL (nb_att_perso%=0) AND (nb_att_ennemi%=0)
RETURN
```

La routine @round_combat utilise la fonction @nb_attaque_perso pour connaître le nombre d'attaques du personnage. Cette fonction assure l'interface entre le programme et une des informations définissant le personnage.

La routine @attaque_ennemi simule une attaque du joueur contre son adversaire. La routine @attaque_perso simule une attaque de l'ennemi contre le personnage.

Le personnage attaque son adversaire

La procédure @attaque_ennemi gère l'attaque du personnage contre son ennemi. La probabilité que le personnage touche son adversaire est la compétence de combat du personnage diminuée de la compétence de défense de l'ennemi. Si la probabilité de toucher est

trop faible, elle est ajustée à 5% afin qu'un combattant trop faible aie quand même une chance de toucher son adversaire (même si celui-ci est une créature fabuleusement puissante).

```
' ***************************
' *  LE PERSONNAGE ATTAQUE  *
' *       SON ENNEMI        *
' ***************************
PROCEDURE attaque_ennemi
  LOCAL combat_perso%
  LOCAL defense_creature%
  LOCAL pourcentage%
  LOCAL m$
  '
  combat_perso%=@combat_perso
  defense_creature%=@defense_creature(typ_ennemi%)
  pourcentage%=combat_perso%-defense_creature%
  '
  IF pourcentage%<5
    pourcentage%=5
  ENDIF
  '
  IF (RANDOM(100)+1)<=pourcentage%
    dommage%=@dommage_creature(num_ennemi%)
    IF dommage%<>0
      SUB vie_ennemi%,dommage%
      m$=nom_creature$(typ_ennemi%)
      m$=m$+" reçoit "+STR$(dommage%)
      m$=m$+" points de dommage"
      PRINT m$
    ENDIF
  ENDIF
RETURN
```

Si l'ennemi est touché, le programme affiche le nombre de points de dégâts. Le programme teste la valeur des dommages pour éviter d'afficher un dommage de 0 point de dégât.
Cela peut arriver si une créature dotée d'une forte protection reçoit un coup faible (par exemple un coup de canif contre une armure de plaques).

Calcul des dommages causés par le personnage

La fonction @dommage_creature calcule les dégâts fait par le joueur à son ennemi. Ils sont égaux aux dommages causés par l'arme du personnage, plus le bonus de force du personnage, moins la protection de l'ennemi.
Dans certains cas de figure, le dommage peut être négatif. Cela peut notament arriver avec une arme faible et une grosse valeur de protection. Pour éviter cela, les dégâts négatifs sont automatiquement ajustés à 0.

```
FUNCTION dommage_creature(creature%)
  LOCAL force%
  LOCAL arme%
  LOCAL dommage%
  LOCAL protection%
  '
  force%=@force_perso
  arme%=@arme_perso
  protection%=@protection_creature(creature%)
  '
  dommage%=@dommage_arme(arme%)
  ADD dommage%,@bonus_force(force%)
  SUB dommage%,@valeur_protection(protection%)
  IF dommage%<0
    dommage%=0
  ENDIF
  '
  RETURN dommage%
ENDFUNC
```

L'adversaire attaque le personnage

La procédure @attaque_perso gère l'attaque de l'ennemi contre le personnage. Elle fonctionne de la même manière que la procédure @attaque_ennemi à ceci que l'attaquant et le défenseur sont inversés.

```
' ****************************
' *   L'ENNEMI typ_creature  *
' *   ATTAQUE LE PERSONNAGE  *
' ****************************
PROCEDURE attaque_perso
  LOCAL combat_ennemi%
  LOCAL defense_perso%
  LOCAL pourcentage%
  '
  combat_ennemi%=@combat_creature(typ_ennemi%)
  defense_perso%=@defense_creature(typ_ennemi%)
  pourcentage%=combat_ennemi%-defense_perso%
  '
  IF pourcentage%<5
    pourcentage%=5
  ENDIF
  '
  IF (RANDOM(100)+1)<=pourcentage%
    dommage%=@dommage_perso(typ_ennemi%)
    IF dommage%<>0
      @dec_vie_perso(dommage%)
      m$=nom_perso$+" perd "+STR$(dommage%)
      m$=m$+" points de vie."
      PRINT m$
    ENDIF
  ENDIF
RETURN
```

Cette routine affiche le nombre de points de dégâts encaissés par le personnage quand il est touché. Un test sur cette valeur évite d'afficher un dommage de 0 point.
La procédure @dec_vie_perso(dommage%) permet de gérer facilement la perte de points de vie du personnage.

Calcul des dommages reçus par le personnage

La fonction @dommage_perso(creature%) calcule les dommages causés par la créature creature% contre le personnage. Leur valeur est calculée en fonction de l'arme et du bonus de force de la créature. Le dommage est diminué de la valeur de protection du personnage. Le principe de fonctionnement est le même que celui de la procédure dommage_creature(creature%).

```
' ************************************************
' *  CALCUL DU DOMMAGE FAIT PAR LA CREATURE      *
' *  creature% AU PERSONNAGE                     *
' ************************************************
FUNCTION dommage_perso(creature%)
  LOCAL force%
  LOCAL arme%
  LOCAL dommage%
  LOCAL protection%
  '
  force%=@force_creature(creature%)
  arme%=@arme_creature(creature%)
  protection%=@protection_perso
  '
  dommage%=@dommage_arme(arme%)
  ADD dommage%,@bonus_force(force%)
  SUB dommage%,@valeur_protection(protection%)
  IF dommage%<0
    dommage%=0
  ENDIF
  '
  RETURN dommage%
ENDFUNC
```

Cette routine utilise la fonction @valeur_protection(protection%) qui permet de connaître le nombre de points de protection assuré par l'objet protection%. Les éventuels dommages négatifs sont automatiquement remis à 0.

Perte des points de vie du personnage

La fonction @dec_vie_perso(n%) diminue les points de vie temporaires du personnage. Le nombre de points de vie temporaire ne peuvent descendre en-dessous de 0.

```
PROCEDURE dec_vie_perso(n%)
  SUB def_perso%(3),n%
  IF def_perso%(3)<0
    def_perso%(3)=0
  ENDIF
RETURN
```

Augmentation des points de vie du personnage

La procédure inc_vie_perso(n%) augmente le nombre de points de vie temporaire de n% points. Elle effectue un test pour s'assurer que le nombre de points ne dépasse pas le nombre de points maximum du personnage.

```
PROCEDURE inc_vie_perso(n%)
  ADD def_perso%(3),n%
  IF def_perso%(3)>@vie_max_perso
    def_perso%(3)=@vie_max_perso
  ENDIF
RETURN
```

Soins médicaux

Après avoir perdu des points de vie dans un combat, le personnage peut se rendre chez un soigneur pour reprendre des forces. Afin de simuler les aléas de la réalité, le personnage ne récupère pas un nombre fixe de points de vie, mais un nombre déterminé avec une formule aléatoire. Un soin permet de récupérer 3 points de vie au minimum et 6 points au maximum. Le tirage se fait avec l'instruction RANDOM(4)+3. La formule RANDOM(4) fournit un nombre aléatoire compris entre 0 et 3.

```
PROCEDURE soins_perso
  LOCAL soins%
  '
  IF @vie_perso=@vie_max_perso
    PRINT "Pas besoin de soins"
  ELSE
    soins%=RANDOM(4)+3
    '
    IF (@vie_perso+soins%)>@vie_max_perso
      soins%=@vie_max_perso-@vie_perso
    ENDIF
    '
    @inc_vie_perso(soins%)
    '
    PRINT "Le personnage récupère ";
    PRINT STR$(soins%);
    PRINT " points de vie."
    '
    PRINT "Points de vie perso : ";
    PRINT @vie_perso;"/";@vie_max_perso
    '
  ENDIF
  PRINT
RETURN
```

La procédure @soins_perso soigne le personnage et affiche le résultat des soins. Le programme ajuste le nombre de points de soins de manière à ne pas dépasser le nombre de points de vie maximal.
Les jeux de rôle micro offrent divers méthodes de soins : sorts de guérison, potions magiques de soins, guérisseurs, herbes médicinales, etc. Ces méthodes fonctionnent toute de la même manière, c'est-à-dire en augmentant le nombre de points de vie. Mais ils peuvent varier au niveau de l'efficacité, du prix et de la disponibilité.
Pour varier les plaisirs, vous pouvez introduire dans votre jeu des soins particuliers permettant de guérir des maux spécifiques (poisons, maladies, malédictions, etc.)

Structure du programme complet

La structure du programme complet est la suivante :

```
RESERVE 100000
' Déclaration des tableaux
@init_datas
@main
END
' Code procédure main
' Codes des procédures du programme
' Définition des données en DATA
```

La procédure principale

La procédure @main est une procédure de test qui vous permet de tester le système de combat. C'est une routine 'minimum' qui simule le programme de jeu de rôle appelant le gestionnaire de combat.

```
PROCEDURE main
  LOCAL c$,creat%
  '
  DO
    PRINT "C)ombat"
    PRINT "S)oins perso"
    PRINT "F)in combat"
    '
    DO
      c$=INKEY$
      c$=UPPER$(c$)
      EXIT IF (c$="C") OR (c$="S") OR (c$="F")
    LOOP
    '
    SELECT c$
    CASE "C","c"
      INPUT "Numéro créature ",creat%
      PRINT
      @exec_combat(creat%)
    CASE "S","s"
      @soins_perso
    ENDSELECT
    EXIT IF (c$="F") OR (c$="f")
    EXIT IF @mort_perso
  LOOP
RETURN
```

Améliorations du programme

Le système de combat peut être amélioré de nombreuses manières. Vous pouvez rajouter une caractéristique Dextérité et faire en sorte que la créature ayant la plus forte dextérité frappe en premier.
Le personnage peut affronter plusieurs ennemis à la fois. Les modifications sont assez faciles à faire, mais prennent beaucoup de place.
La psychologie des créatures du jeu est très simple puisqu'elles combattent jusqu'à la mort. Ce n'est pas très réaliste. Les créatures devraient avoir la possibilité de fuir lorsqu'elles commencent à perdre le combat. De la même manière, le personnage devrait aussi avoir la possibilité de rompre le combat. La visualisation des combats est sommaire. Il est possible d'ajouter des messages supplémentaires afin de rendre les combats plus réalistes. Avec beaucoup de travail, vous pouvez réaliser une visualisation graphique des combats.
Les listes d'objets (armes, protections, trésors et créatures) ne sont pas longues. Il peut être amusant de définir de nouvelles armes et de nouvelles créatures. Si vous nous envoyez des listes intéressantes, nous les mettrons en téléchargement sur le serveur **3615 ATARI**.

3615 ATARI

PLUS DE 1 500 LOGICIELS GRATUITS! TELECHARGEZ

☐ Je commande le KIT DE TELECHARGEMENT (câble de liaison Atari ST/minitel + une disquette contenant des utilitaires de téléchargement) me permettant de télécharger gratuitement des centaines de programmes. De plus la disquette contient le jeu NAVYTEL, pour jouer contre des centaines d'adversaires à travers toute la France grâce au réseau minitel.
Prix exceptionnel:
95 F + 6 F de frais d'expédition*=101 F

☐ Je commande uniquement la disquette du téléchargement, 3 pouces 1/2 pour ST au prix de **15 F** + 6 F de participation aux frais d'expédition=21 F (16 F pour l'étranger=31 F).

☐ Je commande uniquement la disquette du téléchargement, 3 pouces 1/2 pour PC au prix de **16 F** + 6 F de participation aux frais d'expédition=22 F (16 F pour l'étranger=32 F).

NE TAPEZ PLUS LES LISTING! CHARGEZ-LES

☐ Je commande la disquette BEST OF GFA BASIC avec les images au prix de **70 F**.

*Si vous êtes équipé d'un MEGA STE ou d'un TT, le câble de liaison doit être complété par un adaptateur 9/16 broches (non fourni par Artipresse).

Ci-joint mon règlement de ..... F par chèque bancaire ou postal (les mandats ne peuvent être acceptés sauf pour les DOM/TOM et étrangers en FF) à l'ordre d'ARTIPRESSE.

Nom
Prénom
Adresse
Code postal
Ville
Pays
Date de cde 1 9 9

IMPORTANT : il n'est pas accusé réception des commandes. L'encaissement de votre chèque signifie que votre ordre a été enregistré dans le fichier informatique.

Délai d'expédition **3 semaines après encaissement.**

JEU D'AVENTURE A LA SOURIS
Créer un vrai jeu

Voici un petit jeu d'aventure du type Bat ou Final Command fonctionnant entièrement à la souris que vous pourrez modifier et compléter pour obtenir votre propre jeu.

Description du jeu

Un jeu d'aventure est une série de lieux. Le joueur peut se déplacer d'un lieu à un autre, et accomplir diverses actions. Dans ce programme, à chaque lieu est associé une liste de zones d'action. Chaque fois que la souris est placée sur l'une de ces zones, sa forme change. Cette nouvelle forme indique le type d'action associé à cet emplacement de l'écran. Pour exécuter cette action, le joueur n'a plus qu'à cliquer.

Les actions du jeu

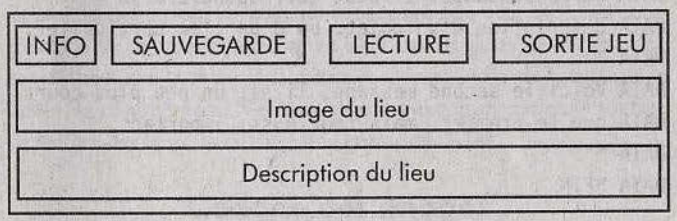
Le jeu proposé ici ne permet pas beaucoup d'actions. C'est un programme de démonstration, et non un jeu commercial. La publication dans un journal a des impératifs draconiens, surtout au niveau de la taille des listing. Les actions existantes sont : changer de lieu, passer une porte, examiner un objet, converser avec un personnage, manger et boire.
Chaque action est associée à une forme de souris particulière, et produit un effet. Le changement de lieu permet de passer d'un endroit à un autre, tout comme le passage d'une porte. L'action 'examen d'un objet' affiche un texte informatif sur l'objet. L'action 'conversation' affiche le dialogue entre le joueur et un personnage de l'aventure. Les actions 'manger' et 'boire' ne produisent aucun effet apparent. Ce sera à vous de les programmer.

Quelques actions supplémentaires

Pour obtenir un jeu vraiment utilisable, il faudrait ajouter beaucoup d'actions. Voici quelques idées : attaquer un personnage, voler un objet, utiliser un appareil électronique quelconque (comme dans *Bat*), acheter divers objets, donner un objet, gérer les objets possédés par le joueur, ouvrir une porte avec une clé (ou défoncer la porte), ouvrir un tiroir, casser une vitre, etc.
Vous pouvez ajouter aussi des options de déplacement comme monter ou descendre un escalier, monter dans un bateau (ou sur un cheval), sauter par une fenêtre, etc.
Les idées d'actions ne manquent pas. Elles dépendent du type d'environnement que vous avez choisis pour votre jeu. Les actions envisageables dans la France des années 1930 ne sont pas les mêmes qu'à Megapolis en 2387 ou qu'à Hangenar, 75000 ans avant notre ère.

Présentation du jeu

L'écran est divisé en trois zones distinctes : les options système, le dessin du lieu courant, et la description textuelle de ce lieu. Les options système sont des fonctions très importantes auxquelles le joueur doit pouvoir accéder en permanence. Ces options sont : l'option "INFO" qui affiche diverses informations sur le logiciel, l'option "SAUVER" qui permet de sauver l'état du jeu sur disquette; l'option "LECTURE" qui permet de relire une sauvegarde et l'option "SORTIE JEU" qui permet de quitter le programme en cours de jeu.



Création d'une aventure

Pour créer une aventure, il faut commencer par étudier le scénario. Un bon scénario, c'est un type d'environnement (fantastique, science-fiction, héroïc-fantasy, historique, etc.), un thème (policier, espionnage, intrigue politique, horreur, etc..), une bonne histoire, des personnages sympathiques ou inquiétants, des lieux originaux (une chapelle gothique, une cité sous-marine, la morgue d'un hôpital spatial du 25ème siècle, la loge d'un concierge, une salle de torture médiévale, un tribunal de l'Inquisition, etc.).

Décomposition de l'aventure

Une fois le scénario terminé, il faut passer par la phase d'analyse et de décomposition. Il faut définir les lieux, les liens existant entre eux, les images les représentant, les textes descriptifs associés à chaque endroit, les personnages, les dialogues et les messages. Ceci

est valable pour le système de jeu que je vous propose ici. Si vous avez rajouté des actions supplémentaires (et je vous le conseille), le travail doit aussi être fait pour ces nouvelles actions.

Codage d'une aventure

Une fois l'aventure décomposée en ses éléments de base, il faut passer à la phase de codage. Le scénario doit être codé dans le programme. La plupart des programmeurs professionnels utilisent des éditeurs de données. Ces éditeurs génèrent des fichiers de données binaires. C'est une technique trop complexe à mettre en oeuvre dans le cadre d'un article. C'est pourquoi j'ai choisi une solution simple, facile à mettre en oeuvre. Les données sont écrites dans des lignes de DATA à la fin du programme source. Au début de l'exécution, une série de routines d'initialisation chargent ces données dans les tableaux de données.

Codage des messages

Les messages sont codés dans des lignes de DATA. Le début de la zone des message est repéré par la présence du label 'messages'. Leur taille n'est pas fixe. Il faut donc un indicateur de fin de message. Cet indicateur est le caractère '*'. L'indicateur de zone de message est '*FIN'.

```
' ************************
' *  EXEMPLES DE MESSAGES  *
' ************************
messages:
DATA Ceci est le premier message du jeu. Il est
DATA affiché lorsque le joueur clique sur la zone
DATA correspondante. Il peut correspondre à la
DATA description d'un objet, ou d'un lieu particulier.
DATA *
DATA Voici le second message. Il est un peu plus court
DATA que le premier, mais tout aussi important.
DATA *
DATA *FIN
```

Codage des dialogues

Les dialogues sont aussi codés dans des lignes de DATA. Leurs structures est identique à celle des messages. Ces dialogues sont très simplifiés. Dans un jeu commercial, il faudrait avoir des échanges verbaux un peu plus étoffés. La zone de stockage de dialogues est indiquée par la présence du label "dialogues".

```
' ************************
' *  EXEMPLES DE DIALOGUES  *
' ************************
dialogues:
DATA Le vieil homme vous répond que la clé de la
DATA sagesse se trouve dans la connaissance, et
DATA non dans une croyance aveugle.
DATA *
DATA Le magicien vous explique qu'il a raison. Il
DATA vous laisse le choix entre être de son avis,
DATA ou recevoir une boule de feu.
DATA *
DATA Le jeune mercenaire vous répond qu'il est du
DATA coté de celui qui lui donne le plus de pièces
DATA d'or.
DATA *
DATA *FIN_DIALOGUES
```

Codage des formes de souris

Les formes de souris sont stockées dans des lignes de DATA selon le format défini dans l'article Bibliothèque de formes de souris. Elles ont été fabriquées à partir d'une image *Degas Elite* grâce au générateur de souris décrit dans l'article sur les souris (source disponible en téléchargement).
Le listing fourni avec cet article comprend 8 formes de souris. Vous pouvez les utiliser pour votre propre programme, ou en rajouter d'autres. Chaque souris a son numéro d'identification. Il n'y a pas une forme de souris pour une action spécifique. Par exemple, les souris <ALLER DROIT DEVANT>, <ENTRER PORTE> et <DEMI-TOUR> correspondent toutes à l'action <DEPLACEMENT>. Pour le joueur, aller devant lui, entrer par une porte ou faire demi-tour dans une rue ne sont pas les mêmes actions, alors que pour le programme, il s'agit toujours de passer dans un autre lieu.

```
1 : <ALLER DROIT DEVANT>
2 : <ENTRER PORTE>
3 : <PARLER>
4 : <MANGER>
5 : <BOIRE>
6 : <FRAPPER>
7 : <OEIL> (pour <EXAMEN>)
8 : <DEMI-TOUR>
```

Codage des actions

Les actions sont des routines du programme. Elles ont toutes un numéro d'identification.

```
Action <DEPLACEMENT> : 1
Action <PARLER>      : 2
Action <MANGER>      : 4
Action <BOIRE>       : 5
Action <EXAMEN>      : 7
```

Codage des lieux

Au niveau interne, les lieux sont définis comme une série d'informations. Ces informations sont : une image, un texte descriptif, et un certain nombre de zones de cliquage correspondant chacune à une action réalisable dans ce lieu.
On associe à chaque action un paramètre spécifiant l'objet avec lesquel l'action doit s'effectuer (le numéro du nouveau lieu pour <DEPLACEMENT>, la personne pour <DIALOGUE>, le numéro de message pour <EXAMEN>, etc..).
L'image est une image dessinée avec le logiciel de dessin *Degas Elite*, stockée sous un format binaire avec le programme GEN_IMGS. La

technique à mettre en oeuvre pour cette opération est expliquée un peu plus bas avec les détails sur GEN_IMGS.
Prenons un exemple de lieu. Il s'agit d'une pièce fermée avec une seule porte. Cette porte donne sur un long couloir. La pièce est éclairée par de grandes torches. Le joueur ne peut accomplir que deux actions: examiner les torches ou passer la porte. La description de ce lieu comprend : le nom du fichier binaire contenant l'image de ce lieu, le texte descriptif de la porte, et la définition des deux zones de clics (les torches et la porte).

```
' **********************
' *  EXEMPLES DE LIEU  *
' **********************
lieux:
' *********************************
' *  LIEU 1 : SALLE AVEC TORCHES  *
' *********************************
DATA SALLE.BIN
DATA Vous vous trouvez dans une grande salle éclairée
DATA par d'immenses torches en métal. La seule issue
DATA visible est une gigantesque porte en bronze.
DATA *
DATA 30,40,100,35,7,7,1
DATA 100,40,30,50,2,1,2
DATA -1
```

La ligne 30,40,50,35,7,7,1 contient le codage de la première zone de clic. Cette zone est positionnée à la position (30,40). Elle à une largeur de 50 pixels, et une hauteur de 35 pixels. Lorsque la souris passe sur cette zone, elle doit avoir la forme 7 (<EXAMEN>). En cas de clic souris cette zone, il faut appeler la routine 7 (<AFFICHAGE D'UN MESSAGE>), et afficher le message numéro 1. Ce message est une description assez horrible des gravures des torches.
La ligne 100,40,30,50,2,1,2 contient le codage de la seconde zone de clic. Cette zone est située à la position (100,40). Elle a une largeur de 30 pixels, et une hauteur de 35 pixels. Lorsque la souris passe dessus, elle doit avoir la forme 2 (<PASSER PORTE>). Si le joueur clique sur cette zone, il faut appeler la routine 1 (<DEPLACEMENT>). Le nouveau lieu sera le lieu 2.
La fin des zones est indiquée par la ligne DATA -1. La fin de tous les lieux est indiquée par la ligne DATA *FIN.

```
' **********************
' *  LIEU 2 : COULOIR  *
' **********************
DATA COULOIR.BIN
DATA Vous vous trouvez actuellement dans un
DATA gigantesque couloir. Il semble s'étendre
DATA à l'infini. Il y a une immense porte
DATA de bronze dans votre dos.
DATA *
DATA 20,80,200,10,8,1,1
DATA -1
DATA *FIN
```

Il n'y a qu'une seule zone de clic dans ce lieu. La forme de souris correspondante est la forme 8 (<DEMI-TOUR>). Lorsque le joueur clique sur cette zone, le programme lance l'exécution de la routine 1 (<DEPLACEMENT>), et le joueur se retrouve dans le lieu 1. La ligne DATA *FIN indique qu'il n'y a pas d'autres lieux dans cette mini-aventure.

Astuce de commentaire

On ne peut pas mettre un commentaire sur la même ligne qu'une instruction DATA, car ce commentaire serait interprété comme une chaîne de caractères appartenant à cette même instruction DATA. C'est ennuyeux, car cela oblige à une répartition inesthétique des commentaires. Pour éviter cela, on peut 'ruser', et mettre un commentaire dans l'instruction DATA comme une chaîne alphanumérique appartenant à l'instruction DATA. Au niveau technique de programmation, c'est une horreur, mais c'est quand même bien plus lisible. La routine de lecture des DATA lit la chaîne de caractère du commentaire, mais ne doit surtout pas en tenir compte.

```
' ***************************************
' *  DEFINITION LIEU AVEC COMMENTAIRE  *
' ***************************************
DATA SALLE.BIN
DATA Vous vous trouvez dans une grande salle
DATA éclairée par d'immenses torches en métal.
DATA La seule issue visible est une gigantesque
DATA porte en bronze.
DATA *
DATA 30,40,100,35,7,7,1,  ! EXAMEN DES TORCHES
DATA 100,40,30,50,2,1,2,  ! ALLER DANS COULOIR
DATA -1
```

Toutes les définitions de zone du programme doivent avoir un commentaire de ce type. Attention : ce n'est pas un véritable commentaire, et il faut toujours avoir le caractère ", entre la fin de la ligne et le début du commentaire. S'il manque un seul commentaire, la routine de lecture des lieux va afficher une erreur de DATA.

Codage des images

Les images sont des blocs graphiques de 308x80 pixels. Elles doivent être dessinées avec un logiciel de dessin style *Degas Elite*. Une fois les images dessinées, elles seront converties dans un format binaire utilisable par le programme de jeu. Cette conversion est faite par le programme CONV_IMG.
La conversion se fait en écrivant dans des lignes de DATA le nom des images *Degas Elite*, la position des dessins sur les images, et le nom des fichiers binaires à générer.
Ces fichiers contiennent la palette de couleur de l'image. Cela permet d'avoir des images avec diverses palettes. Le programme de jeu gère automatiquement les bonnes palettes.

```
' ************************************************
' *  EXEMPLE DE CONVERSION DEGAS/ELITE => BINAIRE  *
' ************************************************
DATA VILLE.PI1
DATA VILLE.BIN
DATA 6,28
```

Cet exemple correspond à la conversion de l'image stockée à la position (6,28) de l'image portant le nom de VILLE.PI1 en un fichier binaire de nom VILLE.BIN. Les fichiers binaires ainsi générés font une taille de 12842 octets.

Générateur d'images Degas Elite

Etant donné qu'il est assez difficile d'insérer des images ou de longs fichiers binaires dans un article, j'ai écrit un programme supplémentaire. Ce petit programme, appelé GEN_IMGS, vous générera des images *Degas elite*. Elles devront être converties par CONV_IMG pour vous donner les fichiers graphiques indispensables au programme de démonstration. La séquence de travail est donc: exécution de GEN_IMGS, exécution de CONV_IMG, puis exécution de DEMO_JEU.

Dossier technique

Le programme a été écrit en *GFA BASIC 3.0*. Sa structure est classique. On y trouve les différents tableaux contenant les informations de jeu, des routines système, un module d'initialisation et des routines spécifiques au programme.

Les données du programme

Les informations du programme sont stockées dans des tableaux de données. Ces derniers sont initialisés à partir des zones de DATA par les routines d'initialisation. Les indices des tableaux commencent à 1 au lieu de 0 grâce à l'instruction OPTION BASE 1 placé au début du programme.
Le tableau message$() contient les divers messages. Le tableau dialogues$() contient le texte des dialogues. Le tableau image$() contient le nom des images associés aux différents lieux. Le tableau descri$() contient les descriptions des lieux.
Le tableau zones%() sert à mémoriser les caractéristiques des zones de clics du jeu. Il y a 7 données par zone de clic, ce tableau a donc deux dimensions (DIM zones%[1000,7]).

```
Codage de la zone n% :

- zones%(n%,1) = Position X;
- zones%(n%,2) = Position Y;
- zones%(n%,3) = Largeur zone (tx);
- zones%(n%,4) = Hauteur zone (ty);
- zones%(n%,5) = Forme de souris;
- zones%(n%,6) = Numéro action;
- zones%(n%,7) = Paramètre action;
```

Les zones sont enregistrées les unes après les autres dans le tableau zones%. Le tableau lieux%() sert à associer les zones aux lieux. La technique de codage est la suivante :

```
- lieux%(n%,1) = Numéro de la première zone du lieu n%
- lieux%(n%,2) = Numéro de la dernière zone du lieu n%
```

La variable sortie% contient la condition d'arrêt du programme. Elle contient habituellement 0, mais si son contenu passe à 1, le programme s'arrête.

Les routines systèmes

Les routines système sont des procédures qui permettent de gérer divers petits problèmes qui se posent tout le temps au programmeur (effacement de l'écran avec une couleur particulière, affichage d'un rectangle graphique, dessin d'un cadre, attente d'un clic souris, etc..). Avoir une bonne bibliothèque de fonctions système permet de réaliser un gain de temps appréciable dans l'écriture de nouveaux programmes.
La routine de formatage de texte est une routine système comme les autres, mais elle nécessite quelques explications. Elle permet de découper un texte quelconque en lignes de textes formatées en respectant les espaces. Les mots ne sont pas coupés à la ligne comme avec un affichage classique (PRINT), et restent donc entiers.

Les routines d'initialisation

Les routines d'initialisation lisent les paramètres du jeu stockés en DATA, et les mettent dans les divers tableaux de données. Ces routines sont @init_messages, @init_dialogues, @init_souris et @init_lieux.
La routine @aff_ecran est aussi une routine d'initialisation, mais au lieu de préparer les données du programme, elle prépare l'écran de jeu qui sera utilisé pendant toute la durée d'exécution du programme.

Les routines de gestion de jeu

La procédure @forme_souris(n%) modifie la forme de la souris. Avec n%=0; la souris a sa forme habituelle de flèche. Avec n%<>0; la souris prend la forme mémorisée dans la variable souris$(n%).
La procédure @att_move_souris attend un changement de position de la souris, ou un clic-souris (changement d'état).
La fonction @tst_zone permet de savoir si une position (x,y) est comprise dans un rectangle graphique (px,py,tx,ty). C'est l'une des fonctions les plus utiles du logiciel.
La fonction @cherche_zone permet de savoir si une position (x,y) de l'écran correspond à l'une des zones de clic du lieu courant. Elle utilise la fonction @tst_zone.
La fonction @cherche_option permet de savoir si une position (x,y) de l'écran correspond à l'une des 4 options du jeu (infos, sauver, lecture, quitter prg). Elle utilise la fonction @tst_zone.
La procédure @att_zone_clic est la routine la plus importante du jeu. C'est elle qui s'occupe de changer la forme de la souris en fonction de sa position, et de contrôler si le joueur clique dans une des zones de l'écran. A chaque changement de position de la souris, elle teste si la souris est sur une zone de clic.
En cas de réponse positive, elle s'occupe de gérer la forme de souris visualisant l'action exécutable à cet endroit de l'écran. Elle utilise les routines @att_move_souris, @cherche_zone, @cherche_option, et @forme_souris.
La procédure @exec_option permet d'exécuter l'une des 4 options du jeu. Elle utilise l'instruction SELECT/CASE pour lancer l'exécution de la routine choisie.
La procédure @exec_action permet d'exécuter l'une des nombreux actions du jeu (déplacement, examen, dialogue, combat, achat, etc.) Chaque action a un numéro. L'instruction SELECT/CASE permet de lancer l'exécution de la procédure correspondant à l'action choisie.

La routine principale du jeu est la procédure @gestion_jeu. Elle attend que l'utilisateur clique sur une zone quelconque, puis appelle la routine qui s'occupe d'exécuter l'action choisie. Elle boucle sur elle-même tant que la variable sortie% contient 0.

Les options systèmes

Ces options sont des fonctions que le joueur peut utiliser en permanence. Il y a la fonction info, la sauvegarde du jeu, la lecture d'une sauvegarde, et une option permettant de sortir du jeu avec demande de confirmation.

Routine d'information

La procédure @option_info affiche une page d'information à l'intention du joueur. Cette page contient le nom du programme, le type de système de développement utilisé (GFA Basic 3.5 en l'occurrence), et les copyrights du programme (l'auteur et Atari Magazine).

Routine de sauvegarde

La procédure @option_sauver sauve sur disquette l'état du jeu. Cet état est très simplifié puisqu'il ne comprend qu'une seule variable : le lieu courant.

Routine de lecture

La procédure @option_lecture lit les données sauvegardées par @option_sauver. Une fois le numéro du lieu courant chargé en mémoire, elle utilise la routine @deplacement pour positionner le jeu dans son nouvel état.

Routine de sortie du programme

La procédure @option_sortie affiche une petite boîte de dialogue avec un texte ('Voulez-vous quitter ce programme ?'), et deux options ('OUI' et 'NON'). Si le joueur clique sur l'option 'NON', il ne se passe rien et le programme quitte la routine. Si le joueur clique sur l'option 'OUI', le programme met 1 dans la variable sortie% avant de quitter la routine.

Les routines d'actions

Ces routines correspondent aux actions exécutables par le joueur. Certaines d'entre elles modifient les variables du programme, d'autres affichent des informations à l'écran, d'autres encore peuvent modifier l'état du joueur (sa position dans le jeu par exemple).
Ces routines ont besoin d'un paramètre décrivant la nature de la tâche à effectuer.

Routine de déplacement

La procédure @deplacement(lieu%) change la position du joueur. C'est une routine importante, puisque l'essence même d'un jeu d'aventure est l'exploration et la découverte de nouveaux endroits. Le changement de lieu se fait en modifiant la variable lieu_courant%, et en affichant l'image et le texte correspondant à ce nouveau lieu.

Routine d'examen

La procédure @affichage_message(num%) affiche le message numéro num%. Le texte est stocké dans le tableau message$(). Ce texte est formaté avec 36 caractères par ligne avec la routine @format avant d'être affiché à l'écran. Une fois le message affiché, le programme affiche une petite boîte de dialogue avec le message 'OK'. Le joueur doit cliquer dans cette boîte pour quitter l'option 'EXAMEN'.

Routine de conversation

La procédure @parler_personnage(perso%) affiche le texte du dialogue numéro perso%. Ce texte est sauvegardé dans le tableau dialogues$(). Une fois le texte affiché, le programme attend que le joueur clique dans une petite boîte 'OK' qu'il dessine après le texte. Ce système de dialogue est simple. Il n'y a pas de véritable conversation. Beaucoup d'anciens jeux utilisent ce principe, mais il est possible de faire mieux et de simuler (du moins partiellement) une véritable conversation.

```
' ****************************************************
' * CONVERSION IMAGES DEGAS ELITE => FORMAT BINAIRE *
' ****************************************************
' ******************************
' *    Programme CONV_IMG.GFA   *
' ******************************
' *  (C) 1990 Patrick Leclercq  *
' *  (C) 1990 ATARI MAGAZINE    *
' ******************************
'
RESERVE 50000
HIDEM
@gen_imgs
SHOWM
END
'
' ____________________________________________
' ****************************************
' *  CHARGEMENT D'UNE IMAGE DEGAS ELITE  *
' ****************************************
' ****************************************************
' *  La palette est stockée dans la variable globale*
' *  palette$. Cette variable est réutilisée par la *
' *  procédure @sauve_image().                      *
' ****************************************************
PROCEDURE load_degas(nom$)
  palette$=SPACE$(32) ! Réservation mémoire 32 octets
  OPEN "i",#99,nom$        ! Ouverture Fichier nom$
  SEEK #99,2  ! ON SAUTE LES DEUX PREMIERS OCTETS
  BGET #99,VARPTR(palette$),32     ! Lecture palette img
  VOID XBIOS(6,L:VARPTR(palette$)) ! Mise à jour palette
  BGET #99,XBIOS(2),32000 ! LECTURE IMAGE SUR ECRAN
  CLOSE #99               ! Fermeture du fichier
RETURN
'
' ____________________________________________
```

```
' ****************************************
' *  SAUVEGARDE D'UNE IMAGE SUR DISQUETTE  *
' ****************************************
PROCEDURE sauve_image(nom_image$)
  LOCAL adr%
  '
  taille%=LEN(image$)            ! Taille image
  OPEN "o",#1,nom_image$         ! Ouverture fichier
  BPUT #1,VARPTR(palette$),32    ! Sauv. Palette image
  BPUT #1,VARPTR(taille%),4      ! Sauv. Taille image
  BPUT #1,VARPTR(image$),taille%! Sauv. Zone mémoire img
  CLOSE #1                       ! Fermeture fichier
RETURN
'
' ____________________________________
' ***********************************
' *  BOUCLE DE LECTURE DES IMAGES  *
' ***********************************
' ******************************************************
' * L'image lue est stockée dans la variable globale  *
' * image$.                                            *
' ******************************************************
PROCEDURE gen_imgs
  LOCAL nom_degas$
  LOCAL nom_fichier$
  LOCAL px%,py%
  LOCAL px2%,py2%
  '
  RESTORE donnees_imgs           ! Fixe pointeur DATA
  DO
    READ nom_degas$              ! Lecture nom image DEGAS
    EXIT IF nom_degas$="*FIN"    ! Test si Fin de données
    READ nom_fichier$            ! Lecture nom fichier img
    READ px%,py%                 ! Lecture position image
    px2%=px%+307                 ! Calcul px2%
    py2%=py%+79                  ! Calcul py2%
    @load_degas(nom_degas$)      ! Chargement image DEGAS
    GET px%,py%,px2%,py2%,image$! Lecture image
    @sauve_image(nom_fichier$)   ! Sauv. image sur disque
  LOOP
RETURN
'
' ____________________________________
' ****************************************
' *  STOCKAGE DES COORDONNEES DES IMAGES  *
' ****************************************
' DATA Nom image DEGAS ELITE
' DATA Nom fichier binaire
' DATA px, py
' DATA *FIN
'
' ____________
donnees_imgs:
'
DATA VILLE.PI1
DATA VILLE.BIN
DATA 6,28
'
DATA TAVERNE.PI1
DATA TAVERNE.BIN
DATA 6,28
'
DATA MARCHE1.PI1
DATA MARCHE1.BIN
DATA 6,28
'
DATA MARCHE2.PI1
DATA MARCHE2.BIN
DATA 6,28
'
DATA GARDE.PI1
DATA GARDE.BIN
DATA 6,28
'
DATA PRISON.PI1
DATA PRISON.BIN
DATA 6,28
'
' ____________________________
' ********************************
' *  INDICATEUR DE FIN D'IMAGES  *
' ********************************
DATA *FIN


' ************************************************
' *  GENERATEUR D'IMAGES POUR DEMONSTRATION JEU  *
' ************************************************
' ******************************
' *    Programme GEN_IMGS.GFA   *
' ******************************
' *  (C) 1990 Patrick Leclercq  *
' *  (C) 1990 Atari Magazine    *
' ******************************
'
RESERVE 10000
@def_couleurs
@gen_images
END
'
' ____________________________________________
' ********************************************
' *  DEFINITION PALETTE DE COULEUR DES IMAGES  *
' ********************************************
PROCEDURE def_couleurs
  LOCAL i%
  LOCAL c%
  '
  RESTORE couleurs
  FOR i%=0 TO 15
    READ c%
    VOID XBIOS(7,i%,c%)
  NEXT i%
RETURN
'
' ____________________________
```

```
' ************************************
' *  PALETTE DE COULEURS DES IMAGES  *
' ************************************
couleurs:
DATA $000,$700,$070,$770,$007,$707,$077,$555
DATA $333,$733,$373,$773,$337,$737,$377,$777
'
' ____________________________________
' ************************************
' *  AFFICHAGE D'UN RECTANGLE PLEIN  *
' ************************************
PROCEDURE rect(px%,py%,tx%,ty%,c%)
  LOCAL px2%,py2%
  '
  px2%=px%+tx%-1
  py2%=py%+ty%-1
  DEFFILL c%
  PBOX px%,py%,px2%,py2%
RETURN
'
' ____________________________________________________
' ****************************************************
' *  SAUVEGARDE D'UNE IMAGE AU FORMAT DEGAS ELITE  *
' ****************************************************
PROCEDURE save_degas(nom$)
  LOCAL palette$
  LOCAL resolution$
  LOCAL adr_pal%
  LOCAL adr_resol%
  LOCAL adr%
  LOCAL i%
  '
  palette$=SPACE$(32)            ! MEMOIRE POUR PALETTE
  resolution$=SPACE$(2)          ! MEMOIRE POUR RESOL.
  adr_pal%=VARPTR(palette$)      ! ADRESSE ZONE PALETTE
  adr_resol%=VARPTR(resolution$)! ADRESSE ZONE RESOL.
  '
  INT{adr_resol%}=XBIOS(4)       ! LECTURE RESOL. ECRAN
  '
  adr%=adr_pal%
  FOR i%=0 TO 15              ! LECTURE PALETTE COUL.
    INT{adr%}=XBIOS(7,i%,-1)! LECTURE COULEUR i%
    ADD adr%,2
  NEXT i%
  '
  OPEN "o",#99,nom$           ! OUVERTURE FICHIER
  BPUT #99,adr_resol%,2       ! ECRITURE RESOLUTION
  BPUT #99,adr_pal%,32        ! ECRITURE PALETTE COUL.
  BPUT #99,XBIOS(2),32000     ! ECRITURE DE L'IMAGE
  CLOSE #99                   ! FERMETURE FICHIER
RETURN
'
' ____________________________
' ****************************
' *  DESSIN DES ZONES DE CLIC  *
' ****************************
PROCEDURE dessin_zones
  LOCAL px%,py%,tx%,ty%
  '
  DO
    READ px%
    EXIT IF px%=-1            ! Test si fin de zones
    READ py%,tx%,ty%
    @rect(px%,py%,tx%,ty%,14) ! Dessin zone de clic
  LOOP
RETURN
'
' ___________________________
' ***************************
' *  GENERATION DES IMAGES  *
' ***************************
PROCEDURE gen_images
  LOCAL nom_image$
  '
  RESTORE donnees
  DO
    READ nom_image$
    EXIT IF nom_image$="*FIN"
    CLS
    @rect(6,28,308,80,5)      ! Dessin fond image
    @dessin_zones             ! Dessin des zones de clic
    @save_degas(nom_image$)
  LOOP
RETURN
'
' ________________________
' ************************
' *  DONNEES DES IMAGES  *
' ************************
donnees:
' ______________
DATA VILLE.PI1
DATA 16,48,60,40
DATA 86,38,60,60
DATA 156,48,60,40
DATA 226,38,8,60
DATA 244,48,60,40
DATA -1
' ______________
DATA TAVERNE.PI1
DATA 38,53,20,30
DATA 90,78,80,20
DATA 202,78,80,20
DATA 90,38,60,25
DATA 202,38,60,25
DATA -1
' ___________
DATA MARCHE1.PI1
DATA 20,91,280,15
DATA 22,33,62,50
DATA 100,33,42,50
DATA 158,33,82,50
DATA 256,33,42,50
DATA -1
' ___________
DATA MARCHE2.PI1
DATA 20,91,280,15
DATA 53,33,40,50
```

```
DATA 140,33,40,50
DATA 227,33,40,50
DATA -1
' ____________
DATA GARDE.PI1
DATA 53,33,40,50
DATA 140,33,40,50
DATA 227,33,40,50
DATA -1
' ____________
DATA PRISON.PI1
DATA 34,74,40,30
DATA 34,34,42,30
DATA 104,34,42,50
DATA 174,34,42,50
DATA 244,34,42,50
DATA -1
DATA *FIN

' ***********************************
' *                                 *
' *    JEU D'AVENTURE GRAPHIQUE     *
' *                                 *
' ***********************************
' *  Programmation GFA BASIC 3.5    *
' *  Testé sur 1040 STF & 1040 STE  *
' *  (C) 1990 Patrick Leclercq      *
' *  (C) 1990 Atari Magazine        *
' ***********************************
'
' ____________________________________________
RESERVE 100000  !RESERVATION MEMOIRE POUR SYSTEME BASIC
OPTION BASE 1   !INDICES DE TABLEAUX COMMENCANT A 1
'
DIM souris$(100) !TABLEAU POUR LES FORMES DE SOURIS
nb_souris%=0     !NOMBRE DE SOURIS DANS TABLEAU souris$
'
DIM lieux%(100,2) !TABLEAU POUR LES LIEUX
DIM images$(100)  !Tableau pour les images de lieux
DIM descri$(100)  !Tableau pour les descript. de lieux
DIM zones%(1000,7)!Tableau pour les nb de zones par
lieu
'
DIM messages$(100) ! Tableau pour les messages
nb_messages%=0
'
DIM dialogues$(100)! tableau pour les dialogues
nb_dialogues%=0
'
nb_lieux%=0        ! NOMBRE DE LIEUX EN MEMOIRE
nb_zones%=0        ! NOMBRE DE ZONES EN MEMOIRE
'
DIM lignes$(20) ! Tableau formatage texte pour
affichage
nb_lignes%=0    ! nb de lignes présentes dans ligne$()
'
fond$=SPACE$(32000)! Variable pour mémorisation écran
'
' ________
DEFMOUSE 0          ! FORME SOURIS = ABEILLE
@init_souris        ! INITIALISATION DES FORMES DE SOURIS
@init_lieux         ! INITIALISATION DES LIEUX
@init_messages      ! INITIALISATION DES MESSAGES
@init_dialogues     ! INITIALISATION DES DIALOGUES
lieu_courant%=1
'
' _______
GRAPHMODE (2)       ! MODE D'AFFICHAGE TRANSPARENT
'                   ! POUR AFFICHER LES CARACTERES SUR
'                   ! N'IMPORTE QUEL FOND
' ____________________
HIDEM               ! EFFACEMENT SOURIS
@aff_ecran          ! AFFICHAGE ECRAN DE JEU
@gestion_jeu        ! EXECUTION DU JEU
DEFMOUSE 0          ! FORME SOURIS = FLECHE
END                 ! FIN DU PROGRAMME
' ________________
'
' ***********************************
' *                                 *
' *  ROUTINES SYSTEME DU PROGRAMME  *
' *                                 *
' ***********************************
' ________________________
' ***************************
' *  EFFACEMENT DE L'ECRAN  *
' ***************************
' ****************************
' *  c% : couleur de l'écran  *
' ****************************
PROCEDURE cls_ecran(c%)
  DEFFILL c%            ! Couleur rectangle
  PBOX 0,0,319,199      ! Dessin rectangle
RETURN
'
' ____________________________________
' ************************************
' *  AFFICHAGE D'UN RECTANGLE PLEIN  *
' ************************************
' *  px% : Position x du rectangle   *
' *  py% : Position y du rectangle   *
' *  tx% : Largeur du rectangle      *
' *  ty% : Hauteur du rectangle      *
' *  c%  : Couleur du rectangle      *
' ************************************
PROCEDURE rect(px%,py%,tx%,ty%,c%)
  LOCAL px2%,py2%
  '
  px2%=px%+tx%-1
  py2%=py%+ty%-1
  DEFFILL c%              ! Couleur rectangle
  PBOX px%,py%,px2%,py2%  ! Dessin rectangle
RETURN
'
' ____________________________
' ****************************
```

```
' *  AFFICHAGE D'UN CADRE VIDE  *
' ******************************
' ************************************
' *  px%      : Position x du cadre    *
' *  py%      : Position y du cadre    *
' *  tx%      : Largeur du cadre       *
' *  ty%      : Hauteur du cadre       *
' *  fond%    : Couleur fond du cadre  *
' *  contour% : Couleur bords du cadre *
' ************************************
PROCEDURE cadre(px%,py%,tx%,ty%,fond%,contour%)
  LOCAL px2%,py2%
  '
  px2%=px%+tx%-1
  py2%=py%+ty%-1
  DEFFILL fond%            ! Couleur fond
  PBOX px%,py%,px2%,py2%   ! Dessin fond
  COLOR contour%           ! Couleur contour
  BOX px%,py%,px2%,py2%    ! Dessin contour
RETURN
'
' ____________________________
' ******************************
' *  TEST D'UNE ZONE GRAPHIQUE  *
' ******************************
' ***************************************************
'* Cette fonction teste si la position (x%,y%)
* est comprise dans la zone graphique
* (px%,py%,tx%,ty%). Si la position est comprise
* dans la zone, @tst_zone renvoit 1.
* Dans le cas contraire, elle renvoit 0.
' ***************************************************
FUNCTION tst_zone(x%,y%,px%,py%,tx%,ty%)
  LOCAL px2%,py2%
  LOCAL r%
  '
  px2%=px%+tx%-1
  py2%=py%+ty%-1
  r%=0
  IF (x%>=px%) AND (x%<=px2%)
    IF (y%>=py%) AND (y%<=py2%)
      r%=1
    ENDIF
  ENDIF
  RETURN r%
ENDFUNC
'
' ____________________________
' *******************************
' *  ATTENTE CLIC DANS UNE ZONE  *
' *******************************
' ****************************************************
'*Cette procédure attend que l'utilisateur clique dans*
'*une zone précise de l'écran.                        *
' ****************************************************
PROCEDURE wait_zone(px%,py%,tx%,ty%)
  LOCAL xm%,ym%
  '
  DO
    @att_clic(xm%,ym%)
    EXIT IF @tst_zone(xm%,ym%,px%,py%,tx%,ty%)=1
  LOOP
RETURN
'
' ____________________________________________________
' ****************************************************
' *  ATTENTE CLIC SOURIS AVEC FOURNITURE ETAT SOURIS  *
' ****************************************************
' * Cette procédure attend que le joueur clique avec *
' * la souris. Elle fournit la position (x,y) de clic*
' * à la procédure appellante. Elle ne rend la main  *
' * que lorsque le joueur a relaché le clic souris.  *
' ****************************************************
PROCEDURE att_clic(VAR xm%,ym%)
  LOCAL km%
  '
  DO
    MOUSE xm%,ym%,km%
    EXIT IF km%<>0       ! Att clic souris
  LOOP
  DO
    EXIT IF MOUSEK=0     ! Att arrêt clic
  LOOP
RETURN
'
' ____________________________
' ****************************
' *  ATTENTE CLIC SOURIS  *
' ****************************
' ****************************************************
'*Cette procédure attend que le joueur ait cliqué. Elle*
'*ne rend la main à la procédure appelante que lorsque *
'*le clic a cessé. A la différence de @att_clic, cette *
'*procédure ne donne pas la position de clic.          *
' ****************************************************
PROCEDURE wait_clic
  REPEAT
  UNTIL MOUSEK     ! Attente clic souris
  REPEAT
  UNTIL MOUSEK=0   ! Attente fin de clic
RETURN
'
' ____________________________
' ******************************
' *  ATTENTE ARRET CLIC SOURIS  *
' ******************************
PROCEDURE att0clic
  REPEAT
  UNTIL MOUSEK=0
RETURN
'
' ____________________
' ********************
' *  COULEUR DU TEXTE  *
' ********************
PROCEDURE texte_color(c%)
```

```
  DEFTEXT c%
RETURN
'
' ____________________________________
' ***************************************
' *  AFFICHAGE TEXTE EN CARACTERES GRAS  *
' ***************************************
PROCEDURE texte_gras
  DEFTEXT ,1
RETURN
'
' ____________________________________________
' ********************************************
' *  AFFICHAGE TEXTE EN CARACTERES ITALIQUES  *
' ********************************************
PROCEDURE texte_italique
  DEFTEXT ,4
RETURN
'
' ____________________________________________
' ********************************************
' *  AFFICHAGE TEXTE EN CARACTERES NORMAUX  *
' ********************************************
PROCEDURE texte_normal
  DEFTEXT ,0
RETURN
'
' ____________________________________
' ***************************************
' *  AFFICHAGE TEXTE EN CAR. BRILLANTS  *
' ***************************************
PROCEDURE texte_brillant
  DEFTEXT ,2
RETURN
'
' ____________________________________
' ***************************************
' *  AFFICHAGE TEXTE EN CAR. SOULIGNES  *
' ***************************************
PROCEDURE texte_souligne
  DEFTEXT ,8
RETURN
'
' ____________________________________
' ***************************************
' *  AFFICHAGE TEXTE EN CAR. ENCADRES  *
' ***************************************
PROCEDURE texte_encadre
  DEFTEXT ,16
RETURN
'
' ____________________________
' ****************************
' *  FORMATAGE D'UN TEXTE  *
' ****************************
' ******************************************************
' * Cette routine découpe un texte en plusieurs lignes *
' * pour un affichage propre. Le découpage se fait au  *
' * niveau des mots.                                   *
' * PARAMETRES ROUTINE :                               *
' * ____________________                               *
' * m$      : texte à formater                         *
' * taille% : nombre maximum de lettres par ligne      *
' * SORTIE ROUTINE :                                   *
' * ________________                                   *
' * Les lignes sont stockées dans le tableau lignes$().*
' * La variable nb_lignes% contient le nb de lignes du *
' * tableau lignes$().                                 *
' ******************************************************
PROCEDURE format(m$,taille%)
  LOCAL tampon$
  LOCAL p%
  LOCAL c$
  LOCAL ligne$
  '
  nb_lignes%=0
  tampon$=""
  ligne$=""
  '
  FOR p%=1 TO 20
    lignes$(p%)=""
  NEXT p%
  '
  ' *************************
  ' *  BOUCLE DE DECOUPAGE  *
  ' *************************
  FOR p%=1 TO LEN(m$)
    c$=MID$(m$,p%,1)
    '
    IF c$<>" "
      tampon$=tampon$+c$
    ENDIF
    '
    IF c$=" "
      IF LEN(ligne$)+1+LEN(tampon$)<=taille%
        IF ligne$=""
          ligne$=tampon$
          tampon$=""
        ELSE
          ligne$=ligne$+" "+tampon$
          tampon$=""
        ENDIF
      ELSE
        INC nb_lignes%
        lignes$(nb_lignes%)=ligne$
        ligne$=""
        IF tampon$<>""
          tampon$=tampon$+" "
        ENDIF
      ENDIF
    ENDIF
    '
  NEXT p%
  '
  ' **********************
  ' *  FIN DU DECOUPAGE  *
```

```
' *********************
IF (ligne$="") AND (tampon$<>"")
  INC nb_lignes%
  lignes$(nb_lignes%)=tampon$
  tampon$=""
ENDIF
'
IF (tampon$<>"") AND (ligne$<>"")
  IF LEN(ligne$)+1+LEN(tampon$)<=taille%
    INC nb_lignes%
    lignes$(nb_lignes%)=ligne$+" "+tampon$
    ligne$=""
    tampon$=""
  ELSE
    INC nb_lignes%
    lignes$(nb_lignes%)=ligne$
    ligne$=""
    INC nb_lignes%
    lignes$(nb_lignes%)=tampon$
    tampon$=""
  ENDIF
ENDIF
'
RETURN
'
' ____________________________
' ******************************
' *  AFFICHAGE DU TEXTE FORMATE  *
' ******************************
' ****************************************************
' * Cette routine affiche le texte mis en forme par la *
' * routine @format à partir de la position graphique  *
' * (px%,py%).                                         *
' ****************************************************
PROCEDURE aff_format(px%,py%)
  LOCAL y%
  LOCAL l%
  '
  y%=py%                    ! Pos Y pour début affichage
  FOR l%=1 TO nb_lignes%    ! Boucle affichage message
    TEXT px%,y%,lignes$(l%) ! Affichage texte message
    ADD y%,9                ! Passage ligne suivante
  NEXT l%
RETURN
'
' ____________________________
' ****************************
' *  MEMORISATION DE L'ECRAN  *
' ****************************
' ****************************************************
' *L'écran est mémorisé dans la variable globale fond$*
' ****************************************************
PROCEDURE memorisation_ecran
  SGET fond$
RETURN
'
' ____________________________
' ****************************
' *  RESTAURATION DE L'ECRAN  *
' ****************************
' ****************************************************
' * L'écran à restaurer doit d'abord avoir été stocké *
' * dans la variable globale fond$ par la procédure   *
' * @memorisation_ecran.                              *
' ****************************************************
PROCEDURE restauration_ecran
  SPUT fond$
RETURN
'
' ____________________________
' ****************************
' *  ROUTINES SPECIFIQUES AU PROGRAMME  *
' ****************************
' ____________________________
' ****************************
' *  INITIALISATION DES LIEUX  *
' ****************************
' ****************************************************
' * Cette routine lit les données stockées en DATA, et *
' *  les mémorise dans les tableaux correspondants     *
' *  (images$(), descri$(), zones%(), lieux%()).       *
' ****************************************************
PROCEDURE init_lieux
  LOCAL px%
  LOCAL img$
  LOCAL mess$
  '
  zone_libre%=1
  nb_lieux%=0
  nb_zones%=0
  '
  RESTORE lieux
  DO
    READ img$
    EXIT IF img$="*FIN"
    INC nb_lieux%
    images$(nb_lieux%)=img$
    DO
      READ mess$
      EXIT IF mess$="*"
      mess$=" "+mess$
      descri$(nb_lieux%)=descri$(nb_lieux%)+mess$
    LOOP
    lieux%(nb_lieux%,1)=zone_libre%
    DO
      READ px%
      EXIT IF px%=-1
      zones%(zone_libre%,1)=px%
      READ zones%(zone_libre%,2)
      READ zones%(zone_libre%,3)
      READ zones%(zone_libre%,4)
      READ zones%(zone_libre%,5)
      READ zones%(zone_libre%,6)
      READ zones%(zone_libre%,7)
      READ rem$
      INC nb_zones%
```

```
      INC zone_libre%
    LOOP
    lieux%(nb_lieux%,2)=nb_zones%
  LOOP
RETURN
'
' ____________________________________
' ********************************
' *  INITIALISATION DES MESSAGES  *
' ********************************
' ******************************************************
' *Cette routine lit les messages stockés en DATA, et *
' *les mémorise dans le tableau messages$(). La       *
' *Variable nb_messages% contient le nb de messages.  *
' ******************************************************
PROCEDURE init_messages
  LOCAL m$
  '
  RESTORE messages
  nb_messages%=0
  DO
    READ m$
    EXIT IF m$="*FIN"
    INC nb_messages%
    messages$(nb_messages%)=m$
    DO
      READ m$
      EXIT IF m$="*"
      m$=" "+m$
      messages$(nb_messages%)=messages$(nb_messages%)+m$
    LOOP
  LOOP
RETURN
'
' ____________________________________
' ***********************************
' *  INITIALISATION DES DIALOGUES  *
' ***********************************
' ******************************************************
' *Cette routine lit le texte des dialogues stockés en*
' *DATA, et les mémorise dans le tableau dialogues$().*
' *La variable nb_dialogues% contient le nb dedialogues*
' *de dialogues$().                                   *
' ******************************************************
PROCEDURE init_dialogues
  LOCAL mess$
  LOCAL n%
  '
  RESTORE dialogues
  n%=0
  DO
    READ mess$
    EXIT IF mess$="*FIN_DIALOGUES"
    INC n%
    dialogues$(n%)=mess$
    DO
      READ mess$
      EXIT IF mess$="*"
      mess$=" "+mess$
      dialogues$(n%)=dialogues$(n%)+mess$
    LOOP
  LOOP
  nb_dialogues%=n%
RETURN
'
' ____________________________________
' ************************************
' *  INITIALISATION FORMES DE SOURIS  *
' ************************************
PROCEDURE init_souris
  LOCAL v%,f$
  '
  RESTORE souris
  nb_souris%=0
  DO
    READ v%                 ! Lecture DATA
    EXIT IF v%=-1           ! Test si fin données
    '
    INC nb_souris%          ! Une nouvelle souris
    f$=MKI$(v%)             ! Début données souris
    FOR i%=1 TO 36          ! Début lecture données
      READ v%               ! Lecture DATA souris
      f$=f$+MKI$(v%)        ! Mémorisation données souris
    NEXT i%                 ! Fin Boucle lecture données
    souris$(nb_souris%)=f$! Souris dans souris$()
  LOOP
RETURN
'
' ____________________________________
' ***********************************
' *  RECHERCHE D'UNE ZONE DE CLIC  *
' ***********************************
' ******************************************************
' *Cette routine cherche si la position (xm,ym)       *
' *correspond à une des zones du lieu courant.Si c'est*
' *la cas, fournit le numéro de cette zone. En cas de *
' * recherche négative, elle renvoit la valeur 0.     *
' ******************************************************
' **********************
' *  xm% : Position X  *
' *  ym% : Position Y  *
' **********************
FUNCTION cherche_zone(xm%,ym%)
  LOCAL n%,r%
  LOCAL zone_debut%,zone_fin%
  LOCAL px%,py%,tx%,ty%
  '
  r%=0
  zone_debut%=lieux%(lieu_courant%,1)
  zone_fin%=lieux%(lieu_courant%,2)
  '
  FOR n%=zone_debut% TO zone_fin%
    px%=zones%(n%,1)
    py%=zones%(n%,2)
    tx%=zones%(n%,3)
    ty%=zones%(n%,4)
```

```
    IF @tst_zone(xm%,ym%,px%,py%,tx%,ty%)=1
      r%=n%
    ENDIF
  NEXT n%
  '
  RETURN r%
ENDFUNC
'
' ____________________________________________
' ********************************************
' *  MODIFICATION DE LA FORME DE LA SOURIS  *
' ********************************************
PROCEDURE forme_souris(n%)
  IF n%=0
    DEFMOUSE 0            ! Souris fléche
  ELSE
    DEFMOUSE souris$(n%) ! Souris redéfinis
  ENDIF
RETURN
'
' ____________________________________________
' ********************************************
' *  ATTENTE D'UN DEPLACEMENT DE LA SOURIS  *
' ********************************************
' ****************************************************
' *  Cette routine attend que la souris change de  *
' * position, ou qu'il se produise un clic souris.  *
' ****************************************************
PROCEDURE att_move_souris(VAR xm%,ym%,km%)
  LOCAL x0%,y0%
  '
  x0%=xm%
  y0%=ym%
  DO
    MOUSE xm%,ym%,km%
    EXIT IF ((xm%<>x0%) OR (ym%<>y0%) OR (km%<>0))
  LOOP
RETURN
'
' ____________________________________________
' ********************************************
' *  AFFICHAGE DE LA DESCRIPTION D'UN LIEU  *
' ********************************************
PROCEDURE aff_texte(num_lieux%)
  @format(descri$(num_lieux%),38)!Texte 38 lettres/ligne
  @rect(6,114,308,80,38)          !Efface texte précédent
  @aff_format(8,124)              !Affichage texte formaté
RETURN
'
' ____________________________________________
' ********************************************
' *  AFFICHAGE DE L'IMAGE DU LIEU COURANT  *
' ********************************************
PROCEDURE aff_image(num_lieu%)
  LOCAL palette$
  LOCAL taille%
  LOCAL nom_image$
  '
  nom_image$=images$(num_lieu%)   ! Nom image lieu
  palette$=SPACE$(32)             ! Mémoire pour palette
  '
  OPEN "i",#1,nom_image$          ! Ouverture fichier
  BGET #1,VARPTR(palette$),32     ! Lecture palette image
  BGET #1,VARPTR(taille%),4       ! lecture taille image
  image$=SPACE$(taille%)          ! Mémoire pour image
  BGET #1,VARPTR(image$),taille%  ! Chargement image
  CLOSE #1                        ! Fermeture fichier
  '
  VOID XBIOS(6,L:VARPTR(palette$))! Modif. palette
  PUT 6,28,image$                 ! Aff image
  @aff_texte(num_lieu%)           ! Aff description lieu
RETURN
'
' ____________________________________
' ************************************
' *  AFFICHAGE DE L'ECRAN DE JEU  *
' ************************************
' ******************************************************
' *Cette routine est utilisée au début du programmepour*
' *afficher l'écran de jeu qui sera utilisé tout le    *
' *reste du temps.                                     *
' ******************************************************
PROCEDURE aff_ecran
  @cls_ecran(9)              ! Effacement écran
  @texte_color(0)
  @texte_gras
  @cadre(6,6,56,15,8,0)      ! Cadre INFOS
  TEXT 14,16,"INFOS"
  @cadre(79,6,64,15,8,0)     ! Cadre SAUVER
  TEXT 87,16,"SAUVER"
  @cadre(161,6,72,15,8,0)    ! Cadre LECTURE
  TEXT 169,16,"LECTURE"
  @cadre(249,6,64,15,8,0)    ! Cadre SORTIE
  TEXT 257,16,"SORTIE"
  @texte_normal
  '
  COLOR 0
  BOX 5,27,314,108           ! Cadre du dessin
  BOX 5,113,314,194          ! Cadre description
  '
  @aff_image(lieu_courant%)  ! Aff image de debut
RETURN
'
' ____________________________________
' ************************************
' *  TEST SI CLIC SUR UNE OPTION DU JEU  *
' ************************************
' ******************************************************
' *Cette fonction teste si le joueur a cliqué surl'une*
' *des 4 options du jeu. En cas de recherche positive,*
' *elle renvoit le numéro de l'option cliquée. En cas *
' *de négative, elle renvoit la valeur 0.              *
' ******************************************************
FUNCTION cherche_option(px%,py%)
  LOCAL zone%
  '
```

```
  zone%=0
  ' Test pour option INFO
  IF @tst_zone(px%,py%,6,6,56,15)=1
    zone%=1
  ENDIF
  ' Test pour option SAUVER
  IF @tst_zone(px%,py%,79,6,64,15)=1
    zone%=2
  ENDIF
  ' Test pour option LECTURE
  IF @tst_zone(px%,py%,161,6,72,15)=1
    zone%=3
  ENDIF
  ' Test pour option SORTIE
  IF @tst_zone(px%,py%,249,6,64,15)=1
    zone%=4
  ENDIF
  '
  RETURN zone%
ENDFUNC
'
' ____________________________
' ******************************
' *  ATTENTE CLIC SUR UNE ZONE  *
' ******************************
' ******************************************************
' *Cette procédure attend que l'utilisateur clique sur*
' *une zone de l'écran, ou sur une option joueur. Elle*
' *gére automatiquement les changements de forme de la*
' *souris en fonction des zones de l'écran.            *
' ******************************************************
' * Cette procédure modifie les variables globales     *
' * typ_clic% et zone_clic%.                           *
' * typ_clic% = 1 => clic dans une option utilisateur *
' * typ_clic% = 2 => clic sur zone écran               *
' * zone_clic% = numéro de la zone cliqué              *
' ******************************************************
PROCEDURE att_zone_clic
  LOCAL xm%,ym%,km%
  LOCAL num_souris%
  LOCAL num_zone%
  LOCAL zone_souris%
  LOCAL num_option%
  '
  ' ****************************************
  ' *  DETERMINATION ETAT INITIAL SOURIS  *
  ' ****************************************
  MOUSE xm%,ym%,km%                    !LECTURE ETAT SOURIS
  zone_souris%=@cherche_zone(xm%,ym%)!RECHERCHE NUM ZONE
  IF zone_souris%=0
   num_souris%=0
  ELSE
   num_souris%=zones%(zone_souris%,5)!LECTURE NUM SOURIS
  ENDIF
  @forme_souris(num_souris%)          ! CHANGEMENT SOURIS
  SHOWM                               ! AFFICHAGE SOURIS
  '
  DO
    @att_move_souris(xm%,ym%,km%)! ATT DEPLACEMENT SOURIS
    num_zone%=@cherche_zone(xm%,ym%)! CHERCHE ZONE SOURIS
    IF (num_zone%<>zone_souris%) !TEST CHANGEMENT SI ZONE
      zone_souris%=num_zone%       ! MODIF. ZONE SOURIS
      IF zone_souris%=0
        num_souris%=0
      ELSE
        num_souris%=zones%(num_zone%,5) ! NUM SOURIS
      ENDIF
      @forme_souris(num_souris%)    ! CHANGEMENT SOURIS
    ENDIF
    '
    typ_clic%=0
    IF (km%=1) AND (num_zone%<>0)
      typ_clic%=2
      zone_clic%=num_zone%
    ENDIF
    '
    num_option%=@cherche_option(xm%,ym%)
    IF (km%=1) AND (num_option%<>0)
      typ_clic%=1
      zone_clic%=num_option%
    ENDIF
    EXIT IF typ_clic%<>0
  LOOP
RETURN
'
' ________________________
' ************************
' *  CHANGEMENT DE LIEU  *
' ************************
PROCEDURE deplacement(num_lieu%)
  HIDEM                          ! Effacement souris
  lieu_courant%=num_lieu%        ! Nouveau lieu courant
  @aff_image(lieu_courant%)      ! Aff nouveau lieu
  SHOWM                          ! Aff souris
RETURN
'
' ____________________________
' ****************************
' *  PARLER A UN PERSONNAGE  *
' ****************************
PROCEDURE parler_personnage(perso%)
  @attOclic                     ! Attente arrêt clic
  HIDEM                         ! Effacer souris
  @memorisation_ecran
  @cadre(11,52,298,96,1,0)      ! Aff cadre conversation
  @texte_gras
  TEXT 128,64,"DIALOGUE"        ! Aff titre option
  @texte_normal
  @format(dialogues$(perso%),36)! texte 36 caract./ligne
  @aff_format(16,76)            ! Aff texte dialogue
  '
  @cadre(144,125,32,15,2,0)     ! Cadre pour 'OK'
  @texte_gras
  TEXT 152,135,"OK"             ! Aff de 'OK'
  @texte_normal
  SHOWM
```

```
  @wait_zone(144,125,32,15)     ! Attente clic sur 'OK'
  '
  HIDEM                         ! Effacer souris
  @restauration_ecran
  SHOWM                         ! Afficher souris
RETURN
'
' ____________________________
' ****************************
' *  AFFICHAGE D'UN MESSAGE  *
' ****************************
PROCEDURE affichage_message(num%)
  @attOclic                     ! Attente arrêt clic
  HIDEM                         ! Plus de souris
  @memorisation_ecran
  @cadre(11,52,298,96,1,0)      ! Cadre boite
  @texte_gras
  TEXT 136,64,"EXAMEN"          ! Affichage Titre option
  @texte_normal
  @format(messages$(num%),36)   ! Texte 36 car/ligne
  @aff_format(16,76)            ! Affichage du texte formaté
  '
  @cadre(144,125,32,15,2,0)     ! Cadre boite 'OK'
  @texte_gras
  TEXT 152,135,"OK"
  @texte_normal
  SHOWM                         ! Affichage souris
  '
  @wait_zone(144,125,32,15)     ! Attente clic sur 'OK'
  '
  HIDEM                         ! Remise de l'écran
  @restauration_ecran           ! dans son état
  SHOWM                         ! initial.
RETURN
'
' ___________________________
' ***************************
' *  EXECUTION DES ACTIONS  *
' ***************************
PROCEDURE exec_action(n%,parametre%)
  SELECT n%
  CASE 1
    @deplacement(parametre%)
  CASE 3
    @parler_personnage(parametre%)
  CASE 7
    @affichage_message(parametre%)
  ENDSELECT
RETURN
'
' ________________________
' ************************
' *  OPTION INFORMATION  *
' ************************
PROCEDURE option_info
  @attOclic
  HIDEM
  @memorisation_ecran
  @cadre(43,59,234,82,1,0)
  @texte_gras
  TEXT 66,69,"JEU D'AVENTURE GRAPHIQUE"
  @texte_normal
  TEXT 52,89,"Programmation GFA BASIC 3.0"
  @texte_italique
  TEXT 52,99,"(C) 1990 Patrick Leclercq"
  TEXT 52,109,"(C) 1990 Atari Magazine"
  '
  @cadre(144,117,32,15,2,0)
  @texte_gras
  TEXT 152,127,"OK"
  @texte_normal
  SHOWM
  @wait_zone(144,117,32,15)  ! Attente clic sur 'OK'
  '
  HIDEM
  @restauration_ecran
  SHOWM
RETURN
'
' _______________________
' ***********************
' *  OPTION SAUVER JEU  *
' ***********************
' ******************************************************
' *Cette routine sauve le numéro du lieu courant.Dans*
' *le cadre d'un programme plus gros, on peut sauver *
' *l'état du personnage (argent, points de vie,objets*
' *possédés, etc.)                                    *
' ******************************************************
PROCEDURE option_sauver
  LOCAL sauve$
  '
  @attOclic
  FILESELECT "*.SAV","",sauve$
  IF sauve$<>""
    OPEN "o",#1,sauve$
    PRINT #1,lieu_courant%
    CLOSE #1
  ENDIF
RETURN
'
' ________________________
' ************************
' *  OPTION LECTURE JEU  *
' ************************
PROCEDURE option_lecture
  LOCAL lect$,lieu_sauv%
  '
  @attOclic
  FILESELECT "*.SAV","",lect$
  IF lect$<>""
    OPEN "i",#1,lect$
    INPUT #1,lieu_sauv%
    CLOSE #1
    @deplacement(lieu_sauv%)
  ENDIF
```

```
RETURN
'
' ____________________
' ***********************
' *  OPTION SORTIE JEU  *
' ***********************
' ****************************************************
' *La condition de sortie est la variable sortie%.   *
' *Cette routine affiche deux cadres ('OUI' et 'NON')*
' *et attend que le joueur clique sur l'un d'entre   *
' *eux. Si le clic a eu lieu sur 'OUI', la variable  *
' *sortie% prend la valeur 1 (condition de sortie du *
' *programme).                                       *
' ****************************************************
PROCEDURE option_sortie
  LOCAL xm%,ym%
  LOCAL clic_option%
  '
  @attOclic
  HIDEM
  @memorisation_ecran
  @cadre(39,74,242,52,1,0)
  TEXT 48,84," Voulez-vous vraiment sortir"
  TEXT 48,94,"de ce programme ? "
  @cadre(94,101,40,16,2,0)
  @cadre(188,101,40,16,2,0)
  @texte_gras
  TEXT 102,112,"OUI"
  TEXT 196,112,"NON"
  @texte_normal
  SHOWM
  clic_option%=0
  DO
    @att_clic(xm%,ym%)
    IF @tst_zone(xm%,ym%,94,101,40,16)=1
      clic_option%=1
      sortie%=1
    ENDIF
    IF @tst_zone(xm%,ym%,188,101,40,16)=1
      clic_option%=1
    ENDIF
    EXIT IF clic_option%=1
  LOOP
  HIDEM
  @restauration_ecran
  SHOWM
RETURN
'
' ____________________________________
' ***********************************
' *  EXECUTION DES OPTIONS JOUEUR  *
' ***********************************
'
****************************************************
' *  Cette routine lance l'exécution de la routine   *
' *  associée au numéro de l'option désirée (1 à 4).  *
'
****************************************************
PROCEDURE exec_option(num_option%)
  SELECT num_option%
  CASE 1
    @option_info
  CASE 2
    @option_sauver
  CASE 3
    @option_lecture
  CASE 4
    @option_sortie
  ENDSELECT
RETURN
'
' ____________________
' ********************
' *  GESTION DU JEU  *
' ********************
' ****************************************************
' * Cette routine est la boucle d'exécution du prg.  *
' * Elle contrôle la condition de sortie.            *
' ****************************************************
PROCEDURE gestion_jeu
  LOCAL action%
  LOCAL parametre%
  '
  sortie%=0
  DO
    @att_zone_clic   ! Attente clic joueur
    '
    ' ********************************
    ' *  Test si clic option joueur  *
    ' ********************************
    IF typ_clic%=1
      @exec_option(zone_clic%)
    ENDIF
    '
    ' ***********************************
    ' *  Test si clic dans zone écran  *
    ' ***********************************
    IF typ_clic%=2
      action%=zones%(zone_clic%,6)
      parametre%=zones%(zone_clic%,7)
      @exec_action(action%,parametre%)
    ENDIF
    '
    EXIT IF sortie%=1     ! Condition de sortie du prg
  LOOP
RETURN
'
' ____________________________________________________
' ****************************************************
'          STOCKAGE DES DONNEES DU PROGRAMME         *
' ****************************************************
' ____________________
' ********************
' *  FORMES DE SOURIS  *
' ********************
' FORMAT DE DEFINITION :
```

```
' DATA Position x du point de clic par rapport
'      au coté haut gauche de la souris
' DATA Position y du point de clic par rapport
'      au coté haut gauche de la souris
' DATA mode d'affichage (toujours 1)
' DATA Couleur du masque
' DATA Couleur de la souris
' DATA 16 entiers 16 bits pour le codage du masque
' DATA 16 entiers 16 bits pour le codage de la forme
'
souris:
' ****************************
' *  FORME 1 : ALLER DEVANT  *
' ****************************
DATA 8,8,1,0,1
DATA &0000,&0180,&07E0,&1FF8,&3FFC,&3FFC,&07E0,&0FF0
DATA &0FF0,&1FF8,&1FF8,&3FFC,&3FFC,&7FFE,&7FFE,&0000
DATA &0000,&0000,&0180,&07E0,&1FF8,&07E0,&03C0,&07E0
DATA &07E0,&0FF0,&0FF0,&1FF8,&1FF8,&3FFC,&0000,&0000
'
' ************************************
' *  FORME 2 : ENTRER PAR UNE PORTE  *
' ************************************
DATA 8,8,1,0,1
DATA &0000,&01FF,&01FF,&21FF,&31FF,&39FF,&FDFF,&FFFF
DATA &FFFF,&FDFF,&39FF,&31FF,&21FF,&01FF,&01FF,&0000
DATA &0000,&0000,&00FE,&00FE,&00FE,&10FE,&38FE,&7CBE
DATA &7CFE,&38FE,&10FE,&00FE,&00FE,&00FE,&0000,&0000
'
' **********************
' *  FORME 3 : PARLER  *
' **********************
DATA 8,8,1,0,1
DATA &FE00,&FE3E,&FE7F,&FE7F,&FE7F,&FF7F,&FF7F,&FE7F
DATA &FE7F,&FE3E,&F8E0,&FE00,&FE00,&FE00,&FE00,&FE00
DATA &0000,&0000,&3C3E,&242A,&7C3E,&7E22,&7E3E,&7C12
DATA &7C3E,&7000,&7000,&7000,&7C00,&7C00,&7C00,&0000
'
' **********************
' *  FORME 4 : MANGER  *
' **********************
DATA 8,8,1,0,1
DATA &FE00,&FE00,&FE00,&FE00,&FE00,&FF00,&FF00,&FE00
DATA &FE00,&FAFE,&F8FE,&F8FE,&FAFE,&FE00,&FE00,&FE00
DATA &0000,&0000,&3C00,&2400,&7C00,&7E00,&7E00,&7C00
DATA &7000,&7000,&707C,&707C,&7000,&7000,&7C00,&0000
'
' *********************
' *  FORME 5 : BOIRE  *
' *********************
DATA 8,8,1,0,1
DATA &FE00,&FE78,&FE78,&FE78,&FE78,&FF78,&FF78,&FE78
DATA &FE30,&FE00,&F820,&F810,&F807,&FFFF,&FFFF,&FFFF
DATA &0000,&0000,&3C30,&2430,&7C00,&7E00,&7E00,&7C00
DATA &7C00,&7000,&7000,&7000,&7000,&7002,&7FFE,&0000
'
' ***********************
' *  FORME 6 : FRAPPER  *
' ***********************
DATA 8,8,1,0,1
DATA &0000,&01C0,&1FCE,&1FFE,&1FFC,&1FFC,&7FF8,&7FF8
DATA &7FFC,&7FFE,&7FFE,&3FFE,&3FF8,&3DF0,&30F0,&0000
DATA &0000,&0000,&0180,&0D84,&0FF8,&0FF0,&1FF0,&3FF0
DATA &3FF0,&3FF8,&3FFC,&1FF0,&18E0,&1060,&0000,&0000
'
' **********************************
' *  FORME 7 : OEIL (POUR EXAMEN)  *
' **********************************
DATA 8,8,1,0,1
DATA &FE0,&1FF0,&3FF8,&7FFC,&FFFE,&FFFF,&FFFF,&FFFF
DATA &FFFE,&7FFC,&3FF8,&1FF0,&7C0,&0,&0,&0
DATA &0,&FE0,&1010,&27C8,&5FF4,&BABA,&DBB6,&9C72
DATA &4FE4,&27C8,&1830,&7C0,&0,&0,&0,&0
'
' *************************
' *  FORME 8 : DEMI-TOUR  *
' *************************
DATA 8,8,1,0,1
DATA &0000,&0000,&3FFF,&3FFF,&3FFF,&3FFF,&3FFF,&3FFF
DATA &3F3F,&3F3F,&7FBF,&FFFF,&7FBF,&3F3F,&1E3F,&0C3F
DATA &0000,&0000,&0000,&1FFE,&1FFE,&1FFE,&1FFE,&1E1E
DATA &1E1E,&1E1E,&1E1E,&7F9E,&3F1E,&1E1E,&0C1E,&0000
'
DATA -1
'
' ____________________________
' ********************************
' *                              *
' *  TEXTES DES DIVERS MESSAGES  *
' *                              *
' ********************************
messages:
' ____________________________
' ******************************
' *  MESSAGE 1 : LES POTENCES  *
' ******************************
DATA Les potences sont un triste souvenir de l'époque
de
DATA HELGIR LE BON. Elles rappellent aux étrangers que
la
DATA justice de la cité est rapide et efficace.
DATA *
' ____________________________
' ***************************************
' *  MESSAGE 2 : LES TONNEAUX DE VINS  *
' ***************************************
DATA Un marchand de verroterie tente de convaincre une
DATA jeune femme qu'elle sera plus séduisante avec un
DATA collier de perles de verre. Elle ne semble pas
DATA très convaincue.
DATA *
' ____________________
' ***************************
' *  MESSAGE 3 : ARMURIER  *
' ***************************
DATA Un armurier propose diverses armes. Son stock est
```

```
DATA impressionnant. Il propose un lot de couteaux de
DATA lancer pour un prix très raisonnable.
DATA *
' ____________________________________
' ************************************
' *  MESSAGE 4 : VENDEUR DE BEIGNETS  *
' ************************************
DATA Un vendeur de beignets propose sa marchandise. C'est
DATA appétissant, mais un peu gras.
DATA *
' ____________________________________
' ************************************
' *  MESSAGE 5 : VENDEUR D'HABITS  *
' ************************************
DATA Un jeune marchand à l'air aggressif vend de vieux
DATA habits au prix du neuf.
DATA *
' ____________________________________
' ************************************
' *  MESSAGE 6 : VENDEUR DE CHEVAUX  *
' ************************************
DATA C'est l'endroit idéal pour acheter un cheval. Vous
DATA pouvez y trouver aussi bien un cheval de trait,
DATA qu'un destrier. IL y a même des ânes (cela peut
DATA toujours être utile).
DATA *
' ____________________________________
' ************************************
' *  MESSAGE 7 : EXAMEN MILICIEN  *
' ************************************
DATA Le chef des miliciens est un sergent d'une DATA
quarantaine d'années. Il est très bien armé
DATA et semble compétent. Son aspect guerrier est
DATA contrebalancé par la poupée de chiffons qu'il
DATA fabrique pour sa fille.
DATA *
' ____________________________________
' ************************************
' *  MESSAGE 8 : JOUER AUX DES  *
' ************************************
DATA L'un des joueurs de dés gagne gros. Les autres le
DATA regardent avec une lueur de suspicion dans le
DATA regard.
DATA *
' ____________________________________
' ************************************
' *  MESSAGE 9 : FIN DE L'ESPACE DE JEU  *
' ************************************
DATA La ville s'étend encore plus loin dans cette
DATA direction. Vous ne pouvez pas aller plus loin (pour
DATA l'instant).
DATA *
' ____________________________________
' ************************************
' *  MESSAGE 10 : SALLE DE TORTURE  *
' ************************************
DATA Vous ne pouvez ouvrir cette porte. Le Milicien vous
DATA dit qu'elle mène à la salle de torture. Seul le
DATA bourreau et ses aides ont le droit d'entrer ici (et
DATA les prisonniers).
DATA *
' ____________________________________
' ************************************
' *  INDICATEUR DE FIN DE MESSAGES  *
' ************************************
DATA *FIN
'
' ____________________________________
' ************************************
' *  LES TEXTES DES DIALOGUES  *
' ************************************
dialogues:
' ____________________________________
' ************************************
' *  DIALOGUE 1 : HOMME EN ARME  *
' ************************************
DATA Vous parlez avec l'homme en armes. C'est un prêtre
DATA d'un ordre guerrier. Il vous raconte les exploits
DATA qu'il a vécu avec ses amis aventuriers. Il est très
DATA fier de son bras en métal qui doit être une arme
DATA redoutable.
DATA *
' ____________________________________
' ************************************
' *  DIALOGUE 2 : MENDIANT  *
' ************************************
DATA Vous parlez pendant un moment avec le vieux
DATA mendiant qui se trouve là. Il est heureux car
DATA il a assez de pièces pour se payer du pain blanc.
DATA C'est un jour faste. Les dieux ont été bon avec
DATA lui.
DATA *
' ____________________________________
' ************************************
' *  DIALOGUE 3 : GARDE PRISON  *
' ************************************
DATA Le garde vous raconte qu'il monte la garde toute
DATA la journée. Il est remplacé par deux collègues
DATA pendant la nuit. Il aime ce poste, car c'est
DATA assez calme. Il plaint ses camarades qui doivent
DATA affronter les émeutes de la rue.
DATA *
' ____________________________________
' ************************************
' *  DIALOGUE 4 : PRISONNIER 1  *
' ************************************
DATA Le prisonnier vous explique qu'il a été condamné
DATA à 3 jours de prison pour avoir volé et tué la
DATA belle-mère du chef des Miliciens.
DATA *
' ____________________________________
' ************************************
' *  DIALOGUE 5 : PRISONNIER 2  *
' ************************************
DATA Ce prisonnier attend d'être jugé pour pillage
DATA de tombe, mais il sait déja qu'il sera écartelé
```

```
DATA en place publique. Il est assez content de la vie
DATA qu'il a mené jusqu'ici. Il s'inquiéte un peu pour
DATA l'avenir.
DATA *
' ***********************
' *  FIN DES DIALOGUES  *
' ***********************
DATA *FIN_DIALOGUES
'
' ________________________
' **************************
' *                        *
' *  DEFINITION DES LIEUX  *
' *                        *
' **************************
lieux:
' DATA Nom image
' DATA texte descriptif du lieu
' DATA px,py,tx,ty,souris,action,paramètre, ! REM
' DATA -1
' DATA *FIN
' ____________________________________
' ************************************
' *  LIEU 1 : LA PORTE DE LA CITEE  *
' ************************************
DATA VILLE.BIN
DATA Vous voici devant la porte de la Cité de Randhal où
DATA se trouve l'Epée Runique (d'après la légende). C'est
DATA un endroit où se cotoient toutes les races et toutes
DATA les cultures.
DATA *
DATA 16,48,60,40,2,1,2,      ! ENTRER DANS TAVERNE
DATA 86,38,60,60,1,7,9,      ! MESSAGE DE FIN DU MONDE
DATA 156,48,60,40,1,1,3,     ! ALLER VERS MARCHE
DATA 226,38,8,60,7,7,1,      ! EXAMEN DES POTENCES
DATA 244,48,60,40,2,1,5,     ! ALLER VERS POSTE DE GARDE
DATA -1
' ________________________________________
' ****************************************
' *  LIEU 2 : TAVERNE DE LA LICORNE BLEU  *
' ****************************************
DATA TAVERNE.BIN
DATA Vous êtes dans la Taverne de La Licorne Bleu.
DATA C'est un endroit célèbre sur les 12 mers et les
DATA 8 océans. Quelques marchands jouent au dés dans
DATA un coin de la salle. Un homme en armes est assis
DATA à une petite table.
DATA *
DATA 38,53,20,30,2,1,1,      ! SORTIR DE LA TAVERNE
DATA 90,78,80,20,7,7,8,      ! MESSAGE JOUER AUX DES
DATA 202,78,80,20,3,3,1,     ! PARLER (HOMME EN ARMES)
DATA 90,38,60,25,5,5,1,      ! BOIRE
DATA 202,38,60,25,4,4,1,     ! MANGER
DATA -1
' ________________________
' ************************
' *  LIEU 3 : MARCHE 1  *
' ************************
DATA MARCHE1.BIN
DATA Voici le célèbre marché des 5 peupliers. Cet endroit
DATA est très ancien. La cité a été construite à coté des
DATA entrepôts des premiers marchands. On peut y trouver
DATA à peu près n'importe quoi (à condition d'avoir
DATA beaucoup de pièces d'or).
DATA *
DATA 20,93,280,10,8,1,1, ! DEMI-TOUR VERS DEBUT VILLE
DATA 22,33,62,50,3,3,2,  ! PARLER AU MENDIANT
DATA 100,33,42,50,1,1,4, ! ALLER VERS MARCHE 2
DATA 158,33,82,50,7,7,2, ! MESSAGE POUR ACHAT CAMELOTE
DATA 256,33,42,50,7,7,6, ! VENDEUR DE CHEVAUX
DATA -1
' ________________________
' ************************
' *  LIEU 4 : MARCHE 2  *
' ************************
DATA MARCHE2.BIN
DATA Vous vous trouvez devant d'autres boutiques.
DATA *
DATA 20,91,280,15,8,1,3,  ! DEMI-TOUR VERS MARCHE 1
DATA 53,33,40,50,7,7,3,   ! MESSAGE ARMURIER
DATA 140,33,40,50,7,7,4,  ! MESSAGE VENDEUR DE BEIGNET
DATA 227,33,40,50,7,7,5,  ! MESSAGE VENDEUR D'HABITS
DATA -1
' ____________________________
' ****************************
' *  LIEU 5 : POSTE DE GARDE  *
' ****************************
DATA GARDE.BIN
DATA Vous entrez d'un pas décidé dans le poste de garde
DATA de la Cité. Quelques miliciens s'occupent en
DATA discutant de leurs exploits militaires. La porte
DATA de la prison est ouverte.
DATA *
DATA 53,33,40,50,7,7,7,   ! EXAMINER CHEF MILICIEN
DATA 140,33,40,50,2,1,1,  ! SORTIR DU POSTE DE GARDE
DATA 227,33,40,50,2,1,6,  ! ALLER VERS LA PRISON
DATA -1
' ________________________
' ************************
' *  LIEU 6 : LA PRISON  *
' ************************
DATA PRISON.BIN
DATA Vous êtes dans la prison. C'est un endroit lugubre.
DATA Un Milicien surveille deux prisonniers.
DATA *
DATA 34,74,40,30,2,1,5,      ! ALLER POSTE DE GARDE
DATA 34,34,42,30,3,3,3,      ! PARLER MILICIEN
DATA 104,34,42,50,3,3,4,     ! PARLER PRISONNIER 1
DATA 174,34,42,50,3,3,5,     ! PARLER PRISONNIER 2
DATA 244,34,42,50,2,7,10,    ! SALLE DE TORTURE
DATA -1
' ____________________________________
' ************************************
' *  INDICATEUR DE FIN DE DONNEES  *
' ************************************
DATA *FIN
```

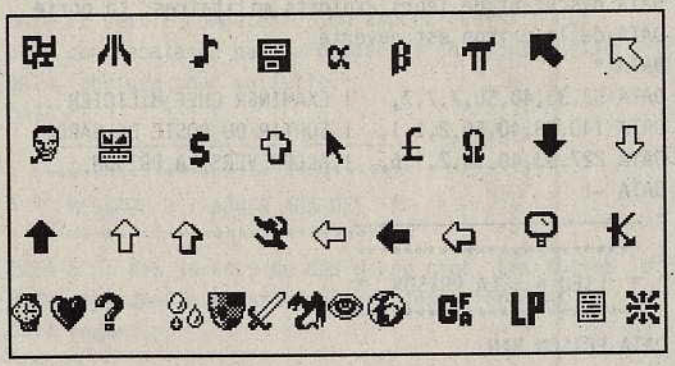
BIBLIOTHEQUE DE FORMES DE SOURIS

Donnez du cachet à vos programmes

Cette bibliothèque offre 40 nouvelles formes de souris à utiliser à partir du GFA Basic. Vous pouvez employer ces souris dans vos propres programmes pour leur donner un air plus sympathique.

Quelques généralités

Les souris mesurent 16 pixels de haut sur 16 pixels de large. Elles ne peuvent être définies qu'en deux couleurs. Le même dessin peut être utilisé dans les 3 résolutions graphiques de l'Atari (basse résolution 320x200 en 16 couleurs, moyenne résolution 640x200 en 4 couleurs, haute résolution 640x400 en deux couleurs et uniquement sur moniteur monochrome). Il est regrettable que les concepteurs du ST n'aient pas prévu de souris adaptées aux différentes résolutions, ce qui aurait permis la création de souris 16 couleurs en basse résolution. Certains jeux utilisent de telles souris, mais pour arriver à leurs fins les programmeurs écrivent entièrement leurs propres routines.



Dessin des souris

Les différentes souris proposées dans cet article ont été dessinées avec le logiciel de dessin *Degas Elite*. Les dessins ont ensuite été transformés en données numériques grâce à un petit programme de conversion écrit en *Basic GFA*. On trouve souvent des éditeurs de souris dans les journaux ou dans les livres, mais il faut savoir que les graphistes professionnels n'utilisent jamais ces utilitaires. Tous les dessins des logiciels commerciaux (animations, formes des souris, polices de caractères, pages de présentation, etc.) sont faits avec des logiciels graphiques classiques, puis récupérés à l'aide de petits programmes astucieux.

Structure d'une souris

Une souris se compose d'une forme, d'un masque de transparence et de deux valeurs de décalage pour les clics «x et y». La forme est le dessin de la souris. Il peut y avoir 2^256 formes différentes, soit 1,16x10^77 souris différentes!

Le masque est un autre dessin indiquant au système les points de la souris qui doivent être affichés à l'écran, et ceux de la souris qui ne doivent pas l'être (c'est-à-dire les points où apparaissent le fond).

Il ne faut pas confondre la souris système avec le dessin qui se déplace sur l'écran. Ce dessin visualise la position de la souris, mais ce n'est pas un point, alors que la souris, elle, ne se trouve qu'à un seul endroit de l'écran.

Lorsque l'utilisateur presse sur un bouton de sa souris, le système détecte le clic et fournit une position (x,y) de l'écran. Cette position est considérée par défaut comme la position du point haut gauche du dessin de la souris.

Les valeurs de décalage x et y permettent de déplacer le véritable «point de clic» par rapport à ce point de clic théorique. En fixant la valeur de décalage en x à 2, le point de clic fourni par le système sera décalé de deux pixels vers la droite.

En prenant de bonnes valeurs de décalage, on peut s'amuser à définir une nouvelle souris dont le clic ne se ferait plus sur la pointe de la flèche mais sur l'autre côté de la souris (certains virus utilisent ce genre d'effet).

Programmation en GFA Basic

L'instruction `Defmouse` (page 176 du manuel *GFA Basic 3.0*) permet de redéfinir la forme de la souris. Les données sont mémorisées dans des lignes de `DATA` sous forme de chiffres hexadécimaux. Cela permet d'avoir des informations faciles à lire et à taper (`&FFB0` se retient plus facilement que 65456 en décimal ou 1111111110110000 en binaire). Les données mémorisées dans les lignes de DATA sont stockées dans un tableau alphanumérique.

La fin des informations est indiquée par la valeur -1 de la dernière ligne de `DATA`. Ce système autorise l'ajout de nouvelles souris sans problème.

```
' ************************************
' *                                  *
' *  BIBLIOTHEQUE DE FORMES DE SOURIS *
' *                                  *
' ************************************
'_____________________________________________
' (c) 1989 ATARI MAGAZINE & Patrick Leclercq
'_____________________________________________
' Machine : ATARI ST
' Langage : BASIC GFA 3.03
' Machine de developpement et de test : ATARI 1040 STF
'_____________________________________________
'
RESERVE 50000        ! RESERVATION MEMOIRE POUR BASIC
'
DIM forme_mouse$(50)! TABLEAU DE FORMES DE SOURIS
nb_mouse%=0          ! NB SOURIS DE TABLEAU forme_mouse$
'
'_____________________________________________
'
@lec_mouse     ! LECTURE DES DEFINITION DE SOURIS
@demo_mouse    ! DEMONSTRATION
END
'
'_____________________________________________
' ***************************************************
' * LECTURE DES FORMES DE SOURIS STOCKES EN DATA    *
' ***************************************************
PROCEDURE lec_mouse
  LOCAL v%,m$
  '
  DO
   READ v%            ! LECTURE DATA
   EXIT IF v%=-1      ! TEST SI FIN DE DONNEES
    '
    INC nb_mouse%     ! UNE NOUVELLE SOURIS
    m$=MKI$(v%)       ! DEBUT DES DONNEES SOURIS
    FOR i%=1 TO 36    ! DEBUT BOUCLE LECTURE DONNEES
     READ v%          ! LECTURE DATA SOURIS
     m$=m$+MKI$(v%)   ! STOCKAGE DES DONNEES SOURIS
    NEXT i%           ! FIN BOUCLE LECTURE DONNEES
    forme_mouse$(nb_mouse%)=m$ ! STOCKAGE SOURIS
  LOOP
RETURN
'
'_____________________________________
' ****************************************
' *  DEMONSTRATION DES FORMES DE SOURIS  *
' ****************************************
PROCEDURE demo_mouse
  LOCAL i%                  ! VARIABLE LOCAL
  '
  FOR i%=1 TO nb_mouse%     ! DEBUT BOUCLE DEMO
   DEFMOUSE forme_mouse$(i%)! AFFICHAGE SOURIS NUMERO
i%
   @attclic                 ! ATTENTE D'UN CLIC SOURIS
  NEXT i%                   ! FIN BOUCLE DEMO
RETURN
'
'_____________________________
' ******************************
' *  ATTENTE D'UN CLIC SOURIS  *
' ******************************
PROCEDURE attclic
  REPEAT                    ! ATTENTE D'UN CLIC SOURIS
  UNTIL MOUSEK
  REPEAT                    ! ATTENTE FIN DU CLIC
SOURIS
  UNTIL MOUSEK=0
RETURN
'
'_____________________________
' ****************************
' *  BIBLIOTHEQUE DE FORMES  *
' ****************************
'
' FORMAT DE DEFINITION :
' DATA Position x du point de clic par rapport
'      au coté haut gauche de la souris
' DATA Position y du point de clic par rapport
'      au coté haut gauche de la souris
' DATA mode d'affichage (1 =SPRITE NORMAL,
' -1 =MODE XOR)
' DATA Couleur du masque
' DATA Couleur de la souris
' DATA 16 entiers 16 bits pour le codage du masque
' DATA 16 entiers 16 bits pour le codage de la forme
'
' REMARQUE : Les souris sont toujours définis en
' 2 couleurs, mais cela peut être n'importe quelles
' couleurs de la palette utilisé par le systéme
' (rouge et vert par exemple). Il n'y a qu'en haute
' résolution (640x400 sur écran monochrome) qu'une
' souris est toujours noir et blanc.
'
montre:
DATA 7,7,1,0,1
DATA &7F8,&FF8,&FF8,&1FFC,&3FFC,&7FFE,&FFFE,&FFFE
DATA &FFFE,&7FFE,&3FFC,&1FFC,&FFC,&FF8,&FF8,&7F8
DATA &3F0,&570,&7F0,&C78,&1118,&292C,&4104,&52F4
DATA &4104,&282C,&1118,&C78,&7F8,&570,&6B0,&3F0
'
coeur:
DATA 7,7,1,0,1
DATA &1C1C,&3F7E,&7FFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &7FFF,&7FFF,&3FFE,&1FFC,&FF8,&7F0,&3E0,&1C0
DATA &0,&1C1C,&3F7E,&67EF,&4FDF,&4FDF,&4FFF,&6FFF
DATA &27FE,&37FE,&1BFC,&DF8,&6F0,&3E0,&1C0,&80
'
point_interrogation:
DATA 7,7,1,0,1
DATA &7F8,&FFC,&1FFE,&3FFF,&3F3F,&3F3F,&1E7F,&FE
DATA &1FC,&3F8,&3F0,&3F0,&3F0,&3F0,&3F0,&3F0
DATA &1E0,&7F8,&BFC,&173E,&161E,&1E1E,&3E,&5C
```

```
DATA &B8,&170,&160,&1E0,&0,&1E0,&160,&1E0
'
larmes:
DATA 7,7,1,0,1
DATA &3C00,&7E00,&FF00,&FF00,&FF90,&FFB8,&FFB8,&FF7C
DATA &7EFE,&3CFE,&39FF,&7DFF,&FFFF,&FEFE,&FE7C,&7C38
DATA &1800,&2400,&4200,&4200,&8500,&8510,&8910,&4228
DATA &3C44,&44,&108A,&288A,&6C92,&4444,&6C38,&3800
'
oeil:
DATA 7,7,1,0,1
DATA &FE0,&1FF0,&3FF8,&7FFC,&FFFE,&FFFF,&FFFF,&FFFF
DATA &FFFE,&7FFC,&3FF8,&1FF0,&7C0,&0,&0,&0
DATA &0,&FE0,&1010,&27C8,&5FF4,&BABA,&DBB6,&9C72
DATA &4FE4,&27C8,&1830,&7C0,&0,&0,&0,&0
'
dragon:
DATA 7,7,1,0,1
DATA &1C3C,&3C3E,&7F7F,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &33FF,&FDF,&1F9F,&3F1F,&3E7F,&3FFF,&3FFF,&1FDA
DATA &818,&181C,&3E3E,&7F3E,&FFFF,&FBFF,&93FF,&31EF
DATA &1CF,&38F,&F0F,&1E0B,&1C0B,&1C72,&1EDA,&F88
'
blason:
DATA 7,7,1,0,1
DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &FFFF,&FFFF,&FFFF,&7FFE,&7FFE,&3FFC,&1FF8,&FF0
DATA &FFFF,&8AFF,&C5FF,&8AFF,&C5FF,&8AFF,&DF55,&EFAB
DATA &5F56,&6FAA,&7756,&26FC,&31FC,&1AF8,&DF0,&3C0
'
epee:
DATA 7,7,1,0,1
DATA &F,&1F,&3F,&7F,&FE,&61FC,&F3F8,&F7F0
DATA &EFE0,&FFC0,&FF80,&7F00,&FEC0,&FFE0,&FFE0,&F7C0
DATA &7,&9,&11,&22,&44,&88,&6110,&E220
DATA &C440,&E880,&7100,&3A00,&6C00,&DEC0,&F7C0,&6380
'
bouclier:
DATA 7,7,1,0,1
DATA &1FF8,&3FFC,&7FFE,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &FFFF,&FFFF,&FFFF,&FFFF,&7FFE,&3FFC,&1FF8,&7E0
DATA &7E0,&18F8,&33A4,&5F82,&4702,&8661,&8FD9,&9F0B
DATA &BE07,&8E0F,&4F8E,&4F9E,&2FBC,&1EF8,&7E0,&0
'
fleche_gauche_pleine:
DATA 1, 1, 1, 0, 1
DATA &180,&380,&780,&F80,&1FFF,&3FFF,&7FFF,&FFFF
DATA &FFFF,&7FFF,&3FFF,&1FFF,&F80,&780,&380,&180
DATA &0,&100,&300,&700,&F00,&1FFE,&3FFE,&7FFE
DATA &7FFE,&3FFE,&1FFE,&F00,&700,&300,&100,&0
'
fleche_gauche_vide:
DATA 1, 1, 1, 0, 1
DATA &0,&300,&780,&F80,&1FFF,&3FFF,&7FFF,&FFFF
DATA &FFFF,&7FFF,&3FFF,&1FFF,&F80,&780,&300,&0
DATA &0,&0,&300,&500,&900,&11FE,&2002,&4002
DATA &4002,&2002,&11FE,&900,&500,&300,&0,&0
'
fleche_gauche_relief:
DATA 1, 1, 1, 0, 1
DATA &0,&300,&780,&F80,&1FFC,&3FFE,&7FFF,&FFFF
DATA &FFFF,&FFFF,&7FFF,&3FDF,&1FCF,&FC0,&7C0,&3C0
DATA &0,&0,&300,&500,&900,&11FC,&2006,&4006
DATA &4006,&6006,&31FE,&19FE,&D80,&780,&380,&0
'
fleche_gauche_haut_vide:
DATA 1, 1, 1, 0, 1
DATA &FFF8,&FFF8,&FFF8,&FFF0,&FFE0,&FFE0,&FFF0,&FFF8
DATA &FFFC,&FFFE,&FFFF,&F3FF,&E1FE,&FC,&78,&30
DATA &0,&7FF0,&4010,&4020,&4040,&4040,&4020,&4010
DATA &4008,&4C04,&5202,&6102,&84,&48,&30,&0
'
fleche_gauche_haut_pleine:
DATA 1, 1, 1, 0, 1
DATA &FFF8,&FFF8,&FFF8,&FFF8,&FFF0,&FFE0,&FFF0,&FFF8
DATA &FFFC,&FFFE,&FFFF,&FBFF,&F1FE,&FC,&78,&30
DATA &0,&7FF0,&7FF0,&7FE0,&7FC0,&7FC0,&7FE0,&7FF0
DATA &7FF8,&7FFC,&73FE,&61FE,&FC,&78,&30,&0
'
fleche_haut_vide:
DATA 1, 1, 1, 0, 1
DATA &180,&3C0,&7E0,&FF0,&1FF8,&3FFC,&7FFE,&7FFE
DATA &3FFC,&FF0,&FF0,&FF0,&FF0,&FF0,&FF0,&FF0
DATA &0,&180,&240,&420,&810,&1008,&2004,&3C3C
DATA &420,&420,&420,&420,&420,&420,&7E0,&0
'
fleche_haut_pleine:
DATA 1, 1, 1, 0, 1
DATA &180,&3C0,&7E0,&FF0,&1FF8,&3FFC,&7FFE,&FFFF
DATA &FFFF,&FF0,&FF0,&FF0,&FF0,&FF0,&FF0,&FF0
DATA &0,&180,&3C0,&7E0,&FF0,&1FF8,&3FFC,&7FFE
DATA &7E0,&7E0,&7E0,&7E0,&7E0,&7E0,&7E0,&0
'
fleche_haut_relief:
DATA 1, 1, 1, 0, 1
DATA &180,&3E0,&7F0,&FF8,&1FFC,&3FFE,&7FFF,&7FFF
DATA &7FFF,&FFF,&FF8,&FF8,&FF8,&FF8,&7F8,&3F8
DATA &0,&180,&260,&430,&818,&100C,&2006,&3C3E
DATA &43E,&430,&430,&430,&430,&7F0,&3F0,&0
'
fleche_bas_vide:
DATA 1, 1, 1, 0, 1
DATA &FF0,&FF0,&FF0,&FF0,&FF0,&FF0,&FF0,&7FFC
DATA &7FFE,&7FFE,&7FFC,&1FF8,&FF0,&7E0,&3C0,&180
DATA &0,&7E0,&420,&420,&420,&420,&420,&420
DATA &3C3C,&2004,&1008,&810,&420,&240,&180,&0
'
fleche_bas_pleine:
DATA 1, 1, 1, 0, 1
DATA &FF0,&FF0,&FF0,&FF0,&FF0,&FF0,&FF0,&FFFF
DATA &FFFF,&7FFE,&3FFC,&1FF8,&FF0,&7E0,&3C0,&180
DATA &0,&7E0,&7E0,&7E0,&7E0,&7E0,&7E0,&7E0
DATA &7FFE,&3FFC,&1FF8,&FF0,&7E0,&3C0,&180,&0
'
```

```
souris_st:
DATA 1, 1, 1, 0, 1
DATA &C000,&E000,&F000,&F800,&FC00,&FE00,&FF00,&FF80
DATA &FFC0,&FF80,&FE00,&EF00,&CF00,&780,&780,&380
DATA &0,&4000,&6000,&7000,&7800,&7C00,&7E00,&7F00
DATA &7F80,&7C00,&6C00,&4600,&600,&300,&300,&0
'
forme_bizarre:
DATA 1, 1, 1, 0, 1
DATA &E038,&F1BC,&FFFE,&7FFF,&7FFF,&3FFF,&1FFF,&27FF
DATA &72FF,&F8FE,&7FFC,&3FFC,&1FFE,&FFF,&7FE,&1E
DATA &0,&6038,&719C,&3FCE,&3FE6,&1FC6,&7EE,&2FE
DATA &207E,&7070,&3870,&1FF8,&FFC,&7FE,&1C,&0
'
loupe:
DATA 1, 1, 1, 0, 1
DATA &1FF8,&3FFC,&7FFE,&F00F,&E087,&E047,&E027,&E007
DATA &E007,&F00F,&7FFE,&3FFC,&1FF8,&7E0,&7E0,&7E0
DATA &0,&1FF8,&3FFC,&700E,&6086,&6046,&6026,&6006
DATA &6006,&700E,&3FFC,&1FF8,&240,&3C0,&240,&3C0
'
transistor:
DATA 1, 1, 1, 0, 1
DATA &3E18,&3E3C,&3E78,&3EF0,&3FE0,&3FC0,&7F80,&FF00
DATA &FF00,&7F80,&3FDC,&3FFC,&3FFC,&3FFC,&3FFC,&3EFC
DATA &0,&1C18,&1C30,&1C60,&1CC0,&1D80,&1F00,&7E00
DATA &7E00,&1F00,&1D80,&1CC8,&1C78,&1C78,&1CF8,&0
'
symbole_etrange:
DATA 1, 1, 1, 0, 1
DATA &FFCF,&FFCF,&FFCF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &FFFF,&FFFF,&FFFF,&FFF8,&FFFE,&FFFE,&FFE,&F80
DATA &0,&7F86,&7F86,&6186,&61FE,&79FE,&7986,&6186
DATA &67FE,&67FE,&6630,&7E30,&7EFC,&6FC,&600,&0
'
symbole_atari:
DATA 1, 1, 1, 0, 1
DATA &7E0,&FF0,&FF0,&FF0,&FF0,&1FF8,&1FF8,&3FFC
DATA &7FFE,&FFFF,&FBDF,&FBDF,&F3CF,&E3C7,&C3C3,&0
DATA &0,&5A0,&5A0,&5A0,&5A0,&DB0,&DB0,&1DB8
DATA &399C,&799E,&718E,&718E,&6186,&4182,&0,&0
'
note_musique:
DATA 1, 1, 1, 0, 1
DATA &1E0,&1E0,&1F8,&1FE,&1FE,&1FE,&1FE,&1EE
DATA &1E0,&1FE0,&1FE0,&7FE0,&7FE0,&7FC0,&1F80,&1F00
DATA &0,&C0,&C0,&F8,&FC,&FC,&C4,&C0
DATA &C0,&C0,&FC0,&FC0,&3FC0,&F00,&F00,&0
'
disquette:
DATA 1, 1, 1, 0, 1
DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &0,&7FFE,&7FFE,&6006,&6FB6,&6006,&6AF6,&6006
DATA &7FFE,&7FFE,&703E,&77BE,&77BE,&703E,&7FFE,&0
'
ordinateur:
DATA 1, 1, 1, 0, 1
DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &0,&7FFE,&4002,&5212,&5A3A,&5E7A,&5EFA,&4002
DATA &7FFE,&240,&7FFE,&4002,&555A,&4002,&7FFE,&0
'
document:
DATA 1, 1, 1, 0, 1
DATA &7FFE,&7FFE,&7FFE,&7FFE,&7FFE,&7FFE,&7FFE,&7FFE
DATA &7FFE,&7FFE,&7FFE,&7FFE,&7FFE,&7FFE,&7FFE,&0
DATA &0,&3FFC,&2004,&2DD4,&2004,&2FD4,&2004,&2BF4
DATA &2004,&2FF4,&2004,&2AE4,&2004,&3FFC,&0,&0
'
personnage:
DATA 1, 1, 1, 0, 1
DATA &FF8,&1FFC,&3FFE,&3FFE,&3FFE,&3FFE,&3FFE,&3FFE
DATA &3FFE,&1FFC,&1FFC,&FFC,&7FF8,&7FF8,&7FF0,&7FE0
DATA &7F0,&FF8,&1FEC,&1804,&1804,&1004,&1E3C,&1754
DATA &1104,&B28,&DD8,&628,&7D0,&2E10,&39E0,&3800
'
tombe:
DATA 1, 1, 1, 0, 1
DATA &7E0,&FF0,&FF8,&3FFC,&7FFE,&7FFF,&7FFF,&7FFF
DATA &7FFF,&3FFF,&FFF,&FF8,&FF8,&FF8,&7F8,&3F0
DATA &0,&7E0,&430,&430,&3C3C,&2006,&2006,&2006
DATA &3C3E,&43E,&430,&430,&430,&7F0,&3F0,&0
'
dollard:
DATA 1, 1, 1, 0, 1
DATA &180,&7F0,&FF8,&FF8,&1FF0,&1FF0,&1FC0,&1FE0
DATA &1FF0,&FF8,&7F8,&FF8,&1FF8,&1FF8,&1FF0,&FC0
DATA &0,&180,&7F0,&7F0,&FF0,&E00,&E00,&FC0
DATA &FC0,&7F0,&70,&70,&FF0,&FF0,&FC0,&180
'
livre:
DATA 1, 1, 1, 0, 1
DATA &1E0,&3F0,&7F0,&FE0,&F00,&F00,&F00,&F00
DATA &3FF0,&3FF0,&3FF0,&F00,&FF8,&3FF8,&3FF8,&3FF8
DATA &0,&1E0,&3E0,&700,&600,&600,&600,&600
DATA &600,&1FE0,&600,&600,&600,&FF0,&1FF0,&0
'
alpha:
DATA 1, 1, 1, 0, 1
DATA &0,&0,&E30,&1FF8,&3FF8,&3FF0,&3FC0,&3FC0
DATA &3FC0,&3FF0,&3FF8,&3FF8,&1F38,&E00,&0,&0
DATA &0,&0,&0,&E30,&1F70,&19C0,&19C0,&1080
DATA &1080,&19C0,&1F70,&1F70,&E30,&0,&0,&0
'
beta:
DATA 1, 1, 1, 0, 1
DATA &1E00,&7F00,&FFC0,&FFC0,&FFC0,&FFC0,&FFC0,&FFC0
DATA &FFC0,&FFC0,&FFC0,&FFC0,&FFC0,&FF00,&FF00,&F800
DATA &0,&C00,&3E00,&7380,&7380,&7380,&7380,&7E00
DATA &7380,&7380,&7380,&7380,&7E00,&7600,&7000,&4000
'
omega:
DATA 1, 1, 1, 0, 1
```

```
DATA &F00,&3F80,&7FC0,&FFE0,&FFE0,&FFE0,&FFE0,&FFE0
DATA &7FC0,&7FC0,&7FC0,&FFE0,&FFE0,&FFE0,&7FC0,&3F80
DATA &0,&F00,&3B80,&70C0,&70C0,&70C0,&70C0,&70C0
DATA &3B80,&3B80,&B00,&4B40,&7BC0,&7BC0,&3B80,&0
'
pi:
DATA 1,1,1,0,1
DATA &6,&F,&1FFF,&3FFF,&7FFF,&FFFE,&FFFC,&FFF8
DATA &FFF0,&6FF0,&FF0,&FF0,&FF0,&FF0,&FF0,&7F0
DATA &0,&6,&6,&1FFE,&1FFE,&7FF8,&7FF8,&6660
DATA &6660,&660,&660,&660,&660,&660,&660,&0
'
symbole_gfa:
DATA 1,1,1,0,1
DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &0,&3E,&3F3E,&7FB0,&7FBC,&73BC,&73B0,&7030
DATA &7780,&779C,&73BE,&7FB6,&7FBE,&3F3E,&36,&0
'
symbole_personnel_auteur:
DATA 1,1,1,0,1
DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &0,&70FE,&70FE,&70EE,&70EE,&70EE,&70EE,&70FE
DATA &70FE,&70FE,&70E0,&70E0,&7EE0,&7EE0,&7EE0,&0
'
symbole_chaotique:
DATA 1,1,1,0,1
DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
DATA &0,&718E,&73CE,&799E,&1DB8,&DB0,&2004,&7E7E
DATA &7E7E,&2004,&DB0,&1DB8,&799E,&73CE,&718E,&0
'
DATA -1


' ****************************************************
' *                                                  *
' *  GENERATION DE FICHIERS CODE POUR FORME DE SOURIS *
' *                                                  *
' ****************************************************
' (c) 1989 Patrick Leclercq & ATARI MAGAZINE
' ****************************************************
' * Cet utilitaire génère un fichier .LST de codage de *
' * souris à partir d'une image DEGAS ELITE contenant  *
' * les formes et les masques des souris. Les positions*
' * des données graphiques sont mémorisés dans des     *
' * lignes de DATA en fin de programme.                *
' ****************************************************
' * La forme de souris et le masque doivent être       *
' * dessinés avec des points de couleur 1 sur un fond  *
' * de couleur 0. Le programme de conversion fonctionne*
' * dans les 3 modes graphiques de l'ATARI.            *
' ****************************************************
' ____________________________________________________
'
RESERVE 50000     ! RESERVATION DE MEMOIRE POUR LE BASIC
@codage           ! APPEL ROUTINE PRINCIPALE
END               ! FIN DU PROGRAMME
'
' ____________________________________________________
' ************************************************
' *  CHARGEMENT D'UNE IMAGE DEGAS SUR L'ECRAN  *
' ************************************************
' ****************************************************
' * Format image DEGAS:                              *
' * - 34 octets contenant la résolution              *
' *   et la palette de couleur;                      *
' * - 32000 octets contenant l'image                 *
' ****************************************************
' *Paramétre: nom$ contient le nom de l'image à charger*
' ****************************************************
PROCEDURE lec_degas(nom$)
 OPEN "I",#1,nom$! OUVERTURE DU FICHIER EN LECTURE
 SEEK #1,34      ! PLACER POINTEUR FICHIER AU DEBUT IMAGE
 BGET #1,XBIOS(2),32000   ! CHARGEMENT IMAGE DANS ECRAN
 CLOSE #1                 ! FERMETURE DU FICHIER
RETURN
'
' ____________________________________________________
' ****************************************************
' *  CODAGE LIGNE DE 16 PIXEL SOUS FORME NUMERIQUE  *
' ****************************************************
' *Paramétres: px% et py% position de la ligne à coder.*
' ****************************************************
FUNCTION lec_ligne(px%,py%)
  LOCAL x%,v%,i%          ! DECLARATION VARIABLES LOCALS
  '
  v%=0                    ! INITIALISATION VALEUR
  x%=px%                  ! INIT POSITION X DE TEST
  FOR i%=15 TO 0 STEP -1  ! BOUCLE DE LECTURE
    c%=POINT(x%,py%)      ! LECTURE COULEUR POINT (x%,y%)
    IF c%<>0              ! TEST SI COULEUR <> 0
      v%=v%+2^i%      ! MISE A JOUR DE LA VALEUR NUMERIQUE
   ENDIF
    INC x%            ! INCREMENTATION POS X DE TEST
   NEXT i%            ! FIN BOUCLE DE LECTURE
  RETURN v%           ! RENVOI DE LA VALEUR NUMERIQUE
ENDFUNC
'
'____________________________________________________
'****************************************************
'* CONVERSION D'UNE FORME 16X16 PIXELS EN CODAGE HEXA *
'****************************************************
'****************************************************
'* Les 16 chiffres hexadécimaux représentant le codage*
'* de la forme sont sont générés sous la forme de deux*
'* lignes de DATA.                                    *
'*DATA &xxxx,&xxxx,&xxxx,&xxxx,&xxxx,&xxxx,&xxxx,&xxxx*
'*DATA &xxxx,&xxxx,&xxxx,&xxxx,&xxxx,&xxxx,&xxxx,&xxxx*
'****************************************************
'* Cette procédure utilise la fonction lec_ligne qui  *
'* convertis une ligne graphique de 16 pixels en un   *
'* nombre entier.                                     *
```

```
'*****************************************************
'* Paramétres de la routine :                        *
'*   px% : position X de la forme à convertir        *
'*   py% : position Y de la forme à convertir        *
'*****************************************************
'* VARIABLES LOCALES :                               *
'*    i% : Indice de boucle (1 à 16)                 *
'*    y% : Position Y de la ligne de pixel à convertir*
'*****************************************************
'* REMARQUE: L'instruction HEX$(v%,4) permet de      *
'* convertir le nombre v% en une chaine hexadécimal de*
'* 4 caractéres (par exemple, la valeur 0 donneras   *
'* la chaine "&0000").                               *
'*****************************************************
PROCEDURE lec_forme(px%,py%)
 LOCAL i%,y%          ! DECLARATION VARIABLES LOCALS
 '
 PRINT #2,"DATA ";    ! INSTRUCTION DATA PREMIERE LIGNE
 '
 y%=py%               ! INITIALISATION POS PREMIÈRE LIGNE
 '
 FOR i%=1 TO 16         ! DEBUT BOUCLE
  v%=@lec_ligne(px%,y%)! LECTURE LIGNE DE POSITION y%
  '
  IF (i%<>8) AND (i%<>16)
    PRINT #2,"&";HEX$(v%,4);","; !ECRITURE FICHIER
  ENDIF
  '
  IF i%=8   ! GENERATION DE LA FIN DE LA PREMIERE LIGNE
    PRINT #2,"&";HEX$(v%,4) ! DERNIERE VALEUR
    PRINT #2,"DATA ";       ! DATA SECONDE LIGNE
  ENDIF
  '
  IF i%=16      ! GENERATION DE LA FIN LIGNE 2
   PRINT #2,"&";HEX$(v%,4) !DERNIERE VALEUR LIGNE 2
   PRINT #2
  ENDIF
  '
  INC y%  ! PASSAGE A LA LIGNE DE PIXEL SUIVANTE
 NEXT i%  ! FIN BOUCLE
RETURN
'
' ____________________________________________
' ************************************
' *  ROUTINE PRINCIPALE DU PROGRAMME  *
' ************************************
' *****************************************************
' *  VARIABLES LOCALS:                                *
' *   image$: Nom de l'image contenant les formes de  *
' *           souris.                                 *
' *   nom$  : Nom du fichier .LST généré par le prg   *
' *           et contenant le codage BASIC des souris.*
' *   px_forme% : Position x de la forme de souris.   *
' *   py_forme% : Position Y de la forme de souris.   *
' *   px_masque%: Position X du masque de la souris.  *
' *   py_masque%: Position Y du masque de la souris.  *
' *****************************************************
PROCEDURE codage
  LOCAL image$,nom$,px_forme%,py_forme%
  LOCAL px_masque%,py_masque%
  '
  READ image$          ! LECTURE NOM IMAGE DEGAS ELITE
  @lec_degas(image$) ! CHARGEMENT DE L'IMAGE
  '
  READ nom$           ! NOM DU FICHIER .LST A ECRIRE
  OPEN "o",#2,nom$  ! OUVERTURE FICHIER .LST
  '
  DO
    READ px_forme%       !LECTURE PREMIERE DONNEE SOURIS
    EXIT IF px_forme%=-1!TEST SI FIN DE DONNEES
    READ py_forme%       !LECTURE POSITION Y FORME SOURIS
    '
    READ px_masque%  ! LECTURE POSITION X MASQUE SOURIS
    READ py_masque%  ! LECTURE POSITION Y MASQUE SOURIS
    '
    PRINT #2,"DATA 1,1,1,0,1" !LIGNE PARAMETRES SOURIS
    '
    lec_forme(px_masque%,py_masque%) ! MASQUE SOURIS
    lec_forme(px_forme%,py_forme%)   ! FORME SOURIS
    PRINT #2,"'"                     ! LIGNE REM
  LOOP
  CLOSE #2            ! FERMETURE DU FICHIER ECRITURE
  '
RETURN
'
' ____________________________________________
' ************************
' *  FORMAT DES DONNEES  *
' ************************
' DATA Nom image DEGAS ELITE
' DATA Nom fichier .LST à générer
' DATA Position X forme souris  1, Pos Y forme souris  1
' DATA Position X masque souris 1, Pos Y masque souris 1
' DATA Position X forme souris  2, Pos Y forme souris  2
' DATA Position X masque souris 2, Pos Y masque souris 2
' DATA            ___            ,          ___
' DATA Position X forme souris  n, Pos Y forme souris  n
' DATA Position X masque souris n, Pos Y masque souris n
' DATA -1
' ____________________________________________
' ***********************************
' *  EXEMPLE DE DONNEES DE SOURIS  *
' ***********************************
DATA SOURIS.PI1
DATA SOURIS.LST
'
DATA 5, 2
DATA 5, 19
'
DATA 22, 2
DATA 22, 19
'
DATA -1
```

LES FONCTIONS GRAPHIQUES DU STE (I)
Maitriser le STE

Cet article en deux parties a été écrit afin de vous montrer comment intégrer ses possibilités graphiques du STE dans vos programmes en GFA Basic.

Palette de couleurs

Les couleurs des STF peuvent être choisies parmi une palette de 512 couleurs tandis que celles des STE peuvent être choisies parmi une palette de 4096 couleurs.

Système RVB

Les couleurs sont fabriquées en mélangeant trois couleurs de bases. Ces couleurs «élémentaires» sont le rouge, le vert et le bleu (d'où le nom de RVB). Chaque couleur élémentaire est divisée en un certain nombre de niveaux de couleurs. Le premier niveau correspond au noir, et le dernier niveau correspond à une couleur très lumineuse.
Le nombre de couleurs intermédiaires dépend du nombre de niveaux de couleurs. Si les trois couleurs de base sont au niveau le plus faible, la couleur résultante est le noir. Par contre, si ces couleurs sont à leur niveau le plus fort, la couleur résultante est le blanc.

De 512 à 4096 couleurs

La palette de couleur du STF est codé sur 3 bits. Cela donne huit niveaux de couleurs (8=2^3). Chacune des trois couleurs élémentaires peut prendre huit teintes différentes. Ce qui représente 8x8x8 (512) couleurs possibles. La définition d'une couleur se fait avec l'instruction SETCOLOR. Cette instruction peut être utilisée de deux manières différentes: en donnant les composantes RVB de manière individuelle, ou en donnant un seul chiffre contenant le codage de la couleur. Chaque technique posséde ses avantages et ses inconvénients.

```
SETCOLOR c,rouge,vert,bleu
SETCOLOR c,mélange
```

La formule permettant de calculer la valeur de la variable mélange est la suivante: `melange=rouge*256+vert*16+bleu`.

On obtient la couleur noire avec les trois composantes RVB à 0 (`SETCOLOR c,0,0,0`). La couleur blanche est obtenue en mettant les trois composantes RVB à leur valeur maximale (`SETCOLOR c,7,7,7`).

Instruction VSETCOLOR

Les routines graphiques du *GFA Basic* ont étés écrites en utilisant les fonctions graphiques internes au ST (système graphique VDI). Ces fonctions sont pratiques et puissantes, mais elles ont un bug très curieux: elles utilisent leur propre table de chiffres pour gérer le numéro des couleurs.
La fonction SETCOLOR utilise le vrai numéro de couleur alors que les autres instructions du *GFA Basic* (COLOR, DEFFILL, etc.) utilisent la table de couleur spécifique au VDI. Pour prendre un exemple précis, si un programme change la définition de la couleur 1 avec SETCOLOR, c'est la couleur 15 qui va être modifiée.

```
SETCOLOR 1,7,0,0
```

L'instruction VSETCOLOR est une instruction qui corrige le bug de couleur. Elle possède sa propre table de conversion et permet de modifier une couleur sans se soucier des problèmes de transcodage de couleur.

```
VSETCOLOR 1,7,0,0
```

Affichage de l'ancienne palette

Le programme suivant affiche successivement toutes les 512 couleurs de la palette STF. Il utilise l'instruction SETCOLOR pour changer les couleurs.

```
' ****************
'* PALETTE STF *
' ****************
FOR rouge%=0 TO 7
 FOR vert%=0 TO 7
  FOR bleu%=0 TO 7
   VSYNC
   SETCOLOR 0,rouge%,vert%,bleu%
  NEXT bleu%
 NEXT vert%
NEXT rouge%
```

Dégradé sur 512 couleurs

Le programme suivant affiche un dégradé de couleur rouge. Pour obtenir un rouge pur, les composantes RVB vert et bleu sont mise à 0. Ce programme affiche huit bandes de couleurs d'une largueur de 320 pixels et d'une hauteur de 8 pixels.

```
FOR i%=0 TO 7
 VSETCOLOR i%,i%,0,0
NEXT i%
'
py%=72
FOR i%=0 TO 7
 DEFFILL i%
 PBOX 0,py%,319,py%+7
 ADD py%,8
NEXT i%
VOID INP(2)
END
```

Palette de couleur étendue du STE

La palette de couleur du STE fonctionne selon le même principe que la palette du STF, mais elle possède une structure légèrement différente. Les composantes RVB sont codées sur 4 bits. Cela représente seize combinaisons différentes (2^4=16). Puisque chaque composante de couleur peut prendre seize valeurs différentes, il peut y avoir 16x16x16 couleurs, c'est-à-dire 4096 couleurs différentes.
Les concepteurs du STE ont réalisé un système de codage des couleurs compatible avec le codage des couleurs STF. C'est-à-dire qu'un codage de couleur STF fonctionnera sur STE. Cette compatibilité est réalisée en rajoutant une nuance intermédiaire entre chaque couleur STF. Celle-ci est codée avec le quatrième bit. Il y a donc huit codages RVB compatibles avec le STF, et huit codages RVB non compatibles STF qui sont des nuances intermédiaires.

Affichage de toutes les couleurs STE

L'affichage des couleurs STE se fait de la même manière que sur le STF, sauf que les codages RVB varient de 0 à 15, et non plus de 0 à 7.

```
FOR rouge%=0 TO 15
 FOR vert%=0 TO 15
  FOR bleu%=0 TO 15
   VSYNC
   SETCOLOR 0,rouge%,vert%,bleu%
  NEXT bleu%
 NEXT vert%
NEXT rouge%
```

Affichage d'un dégradé de couleur

Pour réaliser un dégradé de couleur sur STE, il faut afficher les couleurs dans l'ordre de leurs nuances. A cause du codage particulier du STE, il faut afficher les couleurs dans l'ordre suivant: 0, 8, 1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15.
Le plus simple est d'utiliser l'instruction VSETCOLOR, comme sur STF,pour écrire le programme simplement. Hélas, cela ne marche pas sur les *GFA Basic 3.x* (sauf pour la version *3.5E* qui fonctionne d'une manière un peu particulière). La raison de ce non-fonctionnement nous est inconnue. Le programme suivant vous montre ce que pourrait être une routine d'affichage de dégradés si l'instruction VSETCOLOR fonctionnait correctement sur STE. La séquence des codes RVB est stockée au début du programme dans des lignes de DATA.

```
DATA 0,8,1,9,2,10,3,11
DATA 4,12,5,13,6,14,7,15
'
FOR col%=0 TO 15
 READ rouge%
 VSETCOLOR col%,rouge%,0,0
NEXT col%
'
py%=36
FOR i%=0 TO 15
 DEFFILL i%
 PBOX 0,py%,319,py%+7
 ADD py%,8
NEXT i%
VOID INP(2)
END
```

Pour obtenir un dégradé correct, il faut écrire une routine équivalente à la fonction VSETCOLOR. Pour cela, il faut créer une table numérique qui permet de convertir un numéro de couleur VDI en numéro de couleur réel. Dans le programme d'exemple suivant, La table de conversion est stockée dans le tableau couleur_vdi(). Les données de conversion sont définies dans des lignes de DATA. La routine set_color émule sur un STE l'instruction VSETCOLOR. Attention: pour utiliser correctement set_color, il faut absolument avoir initialisé la table de conversion avec la routine init_couleurs_vdi.

```
' ************************
' *  AFFICHAGE DEGRADE   *
' *  ROUGE SUR STE       *
' ************************
DIM couleur_vdi(16)
'
@init_couleurs_vdi
'
degrade:
DATA 0,8,1,9,2,10,3,11
DATA 4,12,5,13,6,14,7,15
'
couleurs_vdi:
DATA 0,15,1,2,4,6,3,5
DATA 7,8,9,10,12,14,11,13
'
RESTORE degrade
FOR col%=0 TO 15
 READ rouge%
 @set_color(col%,rouge%,0,0)
NEXT col%
```

```
'
py%=36
FOR i%=0 TO 15
 DEFFILL i%
 PBOX 0,py%,319,py%+7
 ADD py%,8
NEXT i%
VOID INP(2)
END
'
' *************************
' *  INITIALISATION TABLE  *
' *  CONVERSION COULEURS   *
' *************************
PROCEDURE init_couleurs_vdi
 LOCAL i%
 RESTORE couleurs_vdi
 FOR i%=0 TO 15
  READ couleur_vdi(i%)
 NEXT i%
RETURN
'
' ************************
' *  ROUTINE EQUIVALENT  *
' *  VSETCOLOR pour STE  *
' ************************
PROCEDURE set_color(nc%,r%,v%,b%)
 LOCAL m%
 m%=r%*256+v%*16+b%
 SETCOLOR couleur_vdi(nc%),m%
RETURN
```

Variations dans les dégradés

Les programmes d'exemples n'affichent que des dégradés de rouge pur, mais vous pouvez afficher aussi des dégradés de vert ou de bleu en modifiant l'appel à la routine set_color.

```
' DEGRADE DE VERT
@set_color(col,0,i%,0)

' DEGRADE DE BLEU
@set_color(col,0,0,i%)
```

En utilisant des mélanges de couleurs, vous pouvez réaliser beaucoup de dégradés différents. Avec la technique présentée ici, il peut y avoir 256 dégradés différents. Ce nombre est obtenu en multipliant le nombre de couleurs fixes possibles (16*16=256).

```
' DEGRADE QUELCONQUE
@set_color(col,2,8,i%)
```

Fabrication de dégradés aléatoires

Ce programme d'exemple affiche une série de dégradés choisie au hasard. Les couleurs constantes sont déterminées aléatoirement avec la fonction RANDOM. La composante RVB de la couleur de dégradé varie de 1 à 15, et non de 0 à 15. Cela permet d'avoir toujours le fond de l'écran en noir.

L'affichage des barres de couleurs est réalisé une fois pour toutes. Le programme affiche un nouveau dégradé en changeant uniquement la palette de couleur grâce à la fonction XBIOS(6). La routine @new_color ne modifie pas directement la palette de couleur courante, mais agit sur une palette stockée dans la zone mémoire d'adresse adrpal. Le programme affiche une nouvelle palette tant que l'utilisateur ne presse pas sur la touche [ESC].

```
' ***************************
' *  AFFICHAGE DE DEGRADES  *
' *        ALEATOIRES       *
' ***************************
DIM col_vdi%(16)
DIM degra%(16)
'
adrpal%=MALLOC(32)
@init_couleurs_vdi
@init_degrade
@calcul_degrade
@aff_degrade
'
DO
 @calcul_degrade
 PAUSE 18
 EXIT IF INP(2)=27
LOOP
VOID MFREE(adrpal%)
END
'
PROCEDURE init_degrade
 LOCAL i%
 '
 RESTORE degrade
 FOR i%=0 TO 15
  READ degra%(i)
 NEXT i%
RETURN
'
' **************************
' * INITIALISATION PALETTE *
' * COULEUR POUR DEGRADE   *
' **************************
PROCEDURE calcul_degrade
 LOCAL v%,b%,r%
 LOCAL alea%
 '
 v%=RANDOM(15)
 b%=RANDOM(15)
 r%=RANDOM(15)
 alea%=RANDOM(3)+1
 '
 @new_color(0,0,0,0)
 FOR i%=1 TO 15
  SELECT alea%
  CASE 1
```

```
  @new_color(i%,r%,v%,degra%(i))
 CASE 2
  @new_color(i%,r%,degra%(i),b%)
 CASE 3
  @new_color(i%,degra%(i),v%,b%)
 ENDSELECT
NEXT i%
VOID XBIOS(6,L:adrpal%)
RETURN
'
' ************************
' *  AFFICHAGE DEGRADE  *
' ************************
PROCEDURE aff_degrade
 LOCAL py%,i%
 '
 py%=36
 FOR i%=0 TO 15
  DEFFILL i%
  PBOX 0,py%,319,py%+7
  ADD py%,8
 NEXT i%
RETURN
'
degrade:
DATA 0,8,1,9,2,10,3,11
DATA 4,12,5,13,6,14,7,15
DATA -1
'
PROCEDURE new_color(nc%,r%,v%,b%)
 LOCAL m%,adr%
 m%=r%*256+v%*16+b%
 adr%=adrpal%+col_vdi%(nc%)*2
 WORD{adr%}=m%
RETURN
'
PROCEDURE init_couleurs_vdi
 LOCAL i%
 RESTORE couleurs_vdi
 FOR i%=0 TO 15
  READ col_vdi(i%)
 NEXT i%
RETURN
couleurs_vdi:
DATA 0,15,1,2,4,6,3,5,7
DATA 8,9,10,12,14,11,13
```

Gestion des couleurs en GFA Basic 3.5E

Le *GFA Basic 3.5E* est une version du *GFA Basic* spécialement adaptée au STE. L'instruction VSETCOLOR ne fonctionne pas de la même manière que les autres version du *GFA Basic*. La séquence des codes RVB à été modifiée. Les codes RVB STE sont maintenant dans l'ordre. Pour afficher un dégradé de couleur, il suffit d'afficher les composantes RVB de 0 à 15.

```
' ***************************
' * AFFICHAGE DEGRADE ROUGE *
' *   AVEC GFA BASIC 3.5E   *
' ***************************
FOR i%=0 TO 15
 VSETCOLOR i%,i%,0,0
NEXT i%
'
py%=36
FOR i%=0 TO 15
 DEFFILL i%
 PBOX 0,py%,319,py%+7
 ADD py%,8
NEXT i%
VOID INP(2)
END
```

Cet exemple ne fonctionnera qu'avec le *Basic GFA 3.5E* alors que l'exemple précédent fonctionnera sur tous les *GFA Basic* de série 3.x (y compris le *GFA Basic 3.5E*).

Scrolling vertical

L'image affichée sur l'écran est la transcription du contenu d'une zone mémoire. Cette image est redessinée 50 fois par seconde. Le dessin est fait par un point lumineux (appelé spot vidéo). Au niveau interne, le circuit électronique d'affichage lit l'adresse de la mémoire écran et affiche les données contenues dans cette zone mémoire sur l'écran. Cette opération s'appelle un rafraîchissement vidéo.
La fonction XBIOS(5) du *GFA Basic* permet de changer l'adresse de la mémoire vidéo. En modifiant cette adresse, on peut réaliser un scrolling très rapide. La mémoire écran de la basse résolution est organisée en ligne de 160 octets. Pour décaler l'écran d'une ligne, il suffit donc de modifier l'adresse de la mémoire vidéo de 160 octets.
Le STE permet de positionner la mémoire vidéo à n'importe quelle adresse de l'écran. Cela permet de réaliser des scrollings très fins. Le STF ne permet de positionner la mémoire écran que sur des adresses multiples de 256. C'est inutilisable pour un scrolling ligne par ligne.

Instruction XBIOS(5)

L'instruction XBIOS(5) est une instruction qui permet de changer l'adresse des écrans physiques et logiques ainsi que la résolution du système. Sa syntaxe est la suivante:

```
VOID XBIOS(5,L:log, L:phys, res)
log  : adresse écran logique
phys : adresse écran physique
res  : résolution écran
```

La résolution ne peut pas être changée en *GFA Basic*, car GEM ne reconnaît pas le changement de résolution. Ce changement ne peut être réalisé qu'avec des programmes entièrement TOS, alors que le *GFA Basic* ne génère que des programmes GEM. Attention: un programme GEM avec une extension .PRG ne devient pas un programme TOS si son extension est renommée en .TOS.
L'adresse de l'écran physique est l'adresse de la mémoire qui sera affichée sur l'écran vidéo, alors que l'adresse de l'écran logique est l'a-

dresse de la zone mémoire où fonctionneront les instructions graphiques du *GFA Basic*. Grâce à ce système d'écran physique et logique, le ST peut très bien afficher une image pendant qu'il est en train d'en dessiner une autre en mémoire. Si un paramétre ne doit pas être modifié, il faut lui donner la valeur -1. L'exemple suivant appelle l'instruction XBIOS(5) sans rien modifier. Cela ne sert à rien, mais c'est un exemple aussi valable qu'un autre.

```
VOID XBIOS(5,L:-1,L:-1,-1)
```

Attention: La modification de l'adresse de la mémoire vidéo n'est pas immédiate. Le système d'affichage ne change l'adresse vidéo que lorsqu'il recommence une phase d'affichage. Et il n'y a que cinquante affichages par seconde. Si un programme modifie plusieurs fois l'adresse de la mémoire vidéo alors que le système est encore en train d'afficher l'ancienne page vidéo, seule la dernière modification sera prise en compte. Pour éviter cela, il faut attendre un réaffichage vidéo après chaque changement de mémoire écran. L'instruction VSYNC est la solution à ce problème: elle bloque l'exécution du programme tant que l'affichage de l'écran n'est pas fini.

```
VOID XBIOS(5,L:-1,L:adr_ecr,-1)
VSYNC
```

Remarque: La syntaxe L:var signifie que le programme doit transmettre une valeur entière 32 bits à la fonction XBIOS. Si vous oubliez le symbole 'L:', vous risquez de gros ennuis, voire même de beaux plantages.

Scrolling vertical pixel par pixel d'une image

Voici un exemple de réalisation pratique du scrolling vertical par modification de la mémoire écran. Le programme crée une zone de travail de 64000 octets. Cette zone est divisée en deux zones mémoire de 32000 octets. Chacune de ces deux zones mémoire a la taille d'une zone écran. Lors de l'initialisation, le programme charge une image au format *Degas Elite* dans chaque zone mémoire. Il déclare ensuite l'adresse travail comme nouvelle mémoire écran. L'image IMG1 apparaît immédiatement à l'écran puisqu'elle se trouve maintenant dans la mémoire écran. Le programme déplace verticalement l'écran d'une ligne en incrémentant l'adresse de la mémoire vidéo de 160 octets. L'utilisateur a l'impression que l'image est montée d'un pixel et qu'une nouvelle ligne est apparue en bas de l'écran. Cette nouvelle ligne est la première ligne de l'image IMG2. En répétant 200 fois cette opération, on va voir l'image IMG1 disparaître vers le haut de l'écran et l'image IMG2 va apparaître à partir du bas de l'écran. Une fois la première boucle terminée, la mémoire écran se trouve alors à l'adresse travail+32000, c'est-à-ire à l'adresse de l'image IMG2. L'image IMG1 a complètement disparu de l'écran. Pour inverser le processus et revenir à l'image IMG1, il faut modifier à nouveau l'adresse de la mémoire vidéo. La seconde partie du programme effectue un nouveau scrolling en décrémentant la mémoire vidéo par tranche de 160 octets. La mémoire vidéo revient à son ancienne valeur. Le spectateur voit l'image IMG2 disparaître vers le bas et l'image IMG1 apparaître à partir du haut de l'écran.

```
' *********************
' *     SCROLLING     *
' *  LIGNE PAR LIGNE  *
' *********************
RESERVE 100000
travail%=MALLOC(64000)
'
adr%=travail%
@load_degas("IMG1.PI1",adr%)
adr%=travail%+32000
@load_degas("IMG2.PI1",adr%)
'
' ****************************
' *  SCROLLING VERS LE BAS   *
' ****************************
adr%=travail%
VOID XBIOS(5,L:-1,L:adr%,-1)
VOID INP(2)
FOR i%=1 TO 200
 ADD adr%,160
 VOID XBIOS(5,L:-1,L:adr%,-1)
 VSYNC
NEXT i%
'
VOID INP(2)
'
' ***************************
' * SCROLLING VERS LE HAUT *
' ***************************
FOR i%=1 TO 200
 SUB adr%,160
 VOID XBIOS(5,L:-1,L:adr%,-1)
 VSYNC
NEXT i%
VOID MFREE(travail%)
END
'
' ************************
' *  CHARGEMENT IMAGE    *
' *  DEGAS ELITE A UNE   *
' *  ADRESSE MEMOIRE     *
' *  QUELCONQUE.         *
' ************************
PROCEDURE load_degas(nom$,adr%)
 LOCAL pal$
 '
 pal$=SPACE$(32)
 OPEN "i",#1,nom$
 BGET #1,VARPTR(pal$),2
 BGET #1,VARPTR(pal$),32
 VOID XBIOS(6,L:VARPTR(pal$))
 BGET #1,adr%,32000
 CLOSE #1
RETURN
```

Attention: après l'exécution de ce programme, la mémoire écran n'est plus à sa valeur initiale. Il faut la réinitialiser pour continuer à travailler normalement. La solution la plus pratique est de sauver l'a-

dresse de la mémoire écran au début du programme afin de la restaurer plus tard.

```
adr_video%=XBIOS(2)
' ......
' Corps du programme
' ......
VOID XBIOS(5,L:-1,L:adr_video%)
```

Utilisation d'un écran virtuel

La technique que nous venons de voir est classique, mais peu facile à gérer. En examinant la situation avec un angle de vision moins «technique», mais plus «pratique», on découvre une autre méthode, plus simple à utiliser.
La zone mémoire de 64000 octets constitue un écran virtuel de 320*400 pixels. La mémoire écran est une fenêtre de 320*200 pixels qui peut être positionnée à n'importe quelle adresse mémoire grâce à la fonction XBIOS(5). On peut réaliser une fonction @set_ligne qui positionne la fenêtre de visualisation sur n'importe quelle ligne de l'écran virtuel. Cette nouvelle fonction calcule l'adresse de la fenêtre de visualisation en multipliant le numéro de la ligne l par 160 (nombre d'octets par ligne en basse résolution). La fonction XBIOS(5) modifie l'adresse de la mémoire vidéo par l'adresse de la fenêtre de visualisation. Attention: l'adresse de l'écran virtuel doit être stockée dans la variable travail.

```
PROCEDURE set_ligne(l%)
 LOCAL adr%
 '
 adr%=travail%+(l%*160)
 VOID XBIOS(5,L:-1,L:adr%,-1)
 VSYNC
RETURN
```

En déplaçant la fenêtre de visualisation sur l'écran virtuel, on réalise un scrolling. L'exemple suivant vous montre la grande simplicité d'utilisation de cette technique.

```
FOR ligne%=0 TO 200
 @set_ligne(ligne%)
NEXT ligne%
```

Pour réaliser un scrolling vers le haut, il suffit de déplacer la fenêtre de visualisation dans le sens interne.

```
FOR ligne%=200 TO 0 STEP -1
 @set_ligne(ligne%)
NEXT ligne%
```

L'exemple suivant réalise un scrolling haut et bas entre 2 images.

```
RESERVE 100000
ravail%=MALLOC(64000)
'
adr%=travail%
@load_degas("IMG1.PI1",adr%)
adr%=travail%+32000
@load_degas("IMG2.PI1",adr%)
'
adr%=travail%
@set_ligne(0)
VOID INP(2)
'
FOR i%=0 TO 200
 @set_ligne(i%)
NEXT i%
'
VOID INP(2)
'
FOR i%=200 TO 0 STEP -1
 @set_ligne(i%)
NEXT i%
'
VOID MFREE(travail%)
END
```

Cette technique peut être utilisée sur des écrans virtuels de n'importe quelle taille. Rien ne vous empêche d'utiliser un écran virtuel de 320*2000 pixels (prenant 320 Ko de mémoire).
Avec une machine équipée de 4 Mo, on peut créer un écran virtuel d'au moins 320x20 000 pixels (3,2 Mo). Avec un STE équipé de 2 Mo de mémoire, nous avons réalisé un scrolling vertical sur un écran virtuel de 320x10000 pixels occupant 1,6 Mégas octets de mémoire. Ce programme a été chargé par 50 images *Degas Elite*. C'est amusant, même si cela ne présente pas beaucoup d'intérêt pratique.

Amélioration de la vitesse de scrolling

Le scrolling pixel par pixel est beau, mais un peu lent. Il est possible de l'améliorer en déplaçant l'écran de plusieurs lignes à la fois. Avec un déplacement de deux lignes à la fois, on augmente la vitesse de scrolling d'un facteur de 2. Avec un déplacement de quatre lignes, on augmente encore la vitesse de scrolling d'un facteur de 2, etc. La qualité visuelle du scrolling n'est pas altérée tant que le pas d'incrémentation est inférieur à 10.

```
FOR ligne%=0 TO 200 STEP 2
 @set_ligne(ligne%)
NEXT ligne%
'
FOR ligne%=200 TO 0 STEP -2
 @set_ligne(ligne%)
NEXT ligne%
```

Bug des boucles FOR-NEXT

Il y a un problème dans les boucles FOR-NEXT avec certaines valeurs d'incrémentation. Ce problème est du au traitement de l'indice de boucle. Avec une boucle simple, la variable l prendra successivement toutes les valeurs entières possibles entre 0 et 200.

```
FOR l%=0 TO 200
 PRINT l%
```

```
NEXT l%
```

Avec un pas d'incrémentation de 2, on obtiendra les valeurs 0, 2, 4, ..., 198, 200.

```
FOR l%=0 TO 200 STEP 2
 PRINT l%
NEXT l%
```

Par contre, la même boucle avec un pas d'incrémentation de 3 donnera un résultat curieux: la variable l% s'arrêtera à la valeur 198. Cela vient de la technique utilisée par le *GFA Basic* pour gérer les boucles FOR-NEXT. Lorsque le programme incrémente la variable compteur de la boucle, il teste si le résultat est supérieur ou égal à la valeur de sortie de boucle. Si ce test est positif, il sort immédiatement de la boucle. Dans notre exemple, la variable de boucle est la variable l%. Lorsque celle-ci vaut 198, le programme l'incrémente de 3. Il teste ensuite le résultat (198+3=201). Comme ce résultat est supérieur à la valeur maximale, il sort de la boucle.

```
FOR l%=0 TO 200 STEP 3
 PRINT l%
NEXT l%
```

Le même phénomène se reproduit avec la valeur 6, car c'est un multiple de 3, ainsi qu'avec la valeur 7. C'est encore plus drôle avec 7, car la valeur de sortie de la boucle est alors 196.
Si vous utilisez un pas d'incrémentation de boucle erroné, le scrolling vertical ne se fera pas sur la totalité de votre écran virtuel. Pour éviter cela, on peut éviter les pas d'incrémentation dangereux ou encore positionner systématiquement la fenêtre de visualisation à sa position théorique de fin de scrolling.

```
FOR l%=0 TO 200 STEP 3
 @set_ligne(l%)
NEXT l%
@set_ligne(200)
```

Pour éviter un affichage supplémentaire, le programme peut faire un test sur la valeur de sortie de la boucle. Cela permet d'utiliser n'importe quel pas de scrolling sans risque. Dans l'exemple suivant, la valeur d'incrémentation de scrolling est stockée dans la variable pas.

```
FOR l%=0 TO 200 STEP pas%
 @set_ligne(l%)
NEXT l%
IF l%<>200
 @set_ligne(200)
ENDIF
```

Remarque: Les problèmes posés par des pas d'incrémentation positifs se retrouvent aussi avec des pas d'incrémentation négatifs. Une boucle FOR-NEXT allant de 200 à 0 avec un pas de -3 finira avec la valeur 2, et la même boucle avec un pas de -7 finira avec la valeur 4.

Scrolling en haute résolution

Les techniques de scrolling vertical que nous venons de voir sont valables pour la basse et moyenne résolution, puisque ces deux résolutions utilisent des lignes de 160 octets. Le cas de la haute résolution est différent car ce mode graphique utilise 400 lignes de 80 octets. Il faut alors modifier la routine @set_ligne_hr.

```
FOR l%=0 TO 400
 @set_ligne_hr(l%)
NEXT l%
'
FOR l%=400 TO 0 STEP -1
 @set_ligne_hr(l%)
NEXT l%
'
PROCEDURE set_ligne_hr(l%)
 LOCAL adr%
 '
 adr%=travail%+(l%*80)
 VOID XBIOS(5,L:-1,L:adr%,-1)
 VSYNC
RETURN
```

Le scrolling en haute résolution est deux fois plus lent que le scrolling en basse ou moyenne résolution, car il y a deux fois plus de lignes à déplacer. Cependant, rien ne vous empêche d'accélérer la vitesse comme nous l'avons fait plus haut.

Scrolling circulaire d'une image

Voici un exemple qui affiche une image scrollant sur elle-même. C'est très spectaculaire, mais facile à programmer. Le programme utilise un écran virtuel de 320*400 pixels (64 000 octets). L'image est chargée en mémoire à partir de la ligne 0 et de la ligne 200. Elle est donc présente deux endroits de la mémoire. Le programme réalise un scrolling vertical entre la ligne 0 et la ligne 200. Une fois ce scrolling terminé, il repositionne la fenêtre de visualisation sur la ligne 0 et recommence le cycle. A tout moment, il visualise sur l'écran une partie de la première image et une partie de la seconde image. C'est le mélange de ces deux images identiques qui donne une impression de scrolling circulaire.
Les images sont chargées aux adresses mémoires adr1 et adr2 Attention: la routine @set_ligne calcule une adresse de ligne à partir de l'adresse adr1 et non travail comme dans les exemples précédents.

```
' *************************
' *  SCROLLING CIRCULAIRE  *
' *      D' UNE IMAGE      *
' *************************
'
RESERVE 100000
adr1%=MALLOC(64000)
adr2%=adr1%+32000
'
@load_degas("IMG1.PI1",adr1%)
@load_degas("IMG1.PI1",adr2%)
'
@set_ligne(0)
'
VOID INP(2)
```

```
DO
 FOR i%=0 TO 200
  @set_ligne(i%)
 NEXT i%
LOOP
'
VOID MFREE(adr1%)
END
'
' ********************
' *  FIXATION LIGNE  *
' *   ECRAN VIRTUEL  *
' ********************
PROCEDURE set_ligne(l%)
 LOCAL adr%
 '
 adr%=adr1%+(l%*160)
 VOID XBIOS(5,L:-1,L:adr%,-1)
 VSYNC
RETURN
```

Ce programme n'a pas de condition d'arrêt. Si vous voulez l'utiliser de manière compilée, il faut prévoir une condition d'arrêt (test avec INKEY, ou utilisation de MOUSEK). On peut inverser le sens du scrolling en inversant le sens de la boucle FOR-NEXT.

```
FOR i%=200 TO 0 STEP -1
 @set_ligne(i%)
NEXT i%
```

Comme dans les exemples précédents, il est possible d'augmenter la vitesse de scrolling en augmentant le pas d'incrémentation entre chaque ligne.

Scrolling vertical commandé par la souris

Le programme suivant déplace une fenêtre de visualisation sur un écran virtuel de 320x400 pixels. La position de la fenêtre de visualisation est commandée par la souris. Une boucle DO-LOOP permet de gérer les événements souris. Le programme ne sort de cette boucle que si la position Y de la souris varie, ou si l'utilisateur clique sur un bouton souris (condition de sortie du programme).
Le résultat est spectaculaire: le scrolling suit tous les mouvements de la souris. Un déplacement lent provoque un scrolling lent et un déplacement rapide modifie très rapidement l'écran.

```
' **************************
' *   SCROLLING VERTICAL   *
' * COMMANDE PAR LA SOURIS *
' **************************
RESERVE 100000
old_ecr%=XBIOS(2)
travail%=MALLOC(64000)
'
HIDEM
adr%=travail%
@load_degas("IMG1.PI1",adr%)
adr%=travail%+32000
@load_degas("IMG2.PI1",adr%)
'
MOUSE xm%,ym0%,km%
@set_ligne(ym0%)
DO
 DO
   MOUSE xm%,ym%,km%
   EXIT IF ym%<>ym0%
   EXIT IF km%<>0
 LOOP
 EXIT IF km%<>0
 ym0%=ym%
 @set_ligne(ym%)
LOOP
'
VOID XBIOS(5,L:-1,L:old_ecr%,-1)
SHOWM
VOID MFREE(travail%)
END
```

Remarque: La position Y de la souris varie entre 0 et 199 alors que la position d'affichage de l'écran virtuel varie entre 0 et 200. La dernière ligne de la seconde image n'est donc jamais affichée sur l'écran.

Scrolling vertical sur un grand écran

On peut réaliser un scrolling sur un écran de grande dimension en fabriquant l'écran au fur et à mesure de l'affichage. L'écran peut être construit avec des éléments graphiques de 16x16 pixels.
L'écran virtuel est constitué d'une série de lignes de blocs de 16x16 pixels. IL y a 320/16 blocs par ligne, c'est-à-dire vingt blocs par ligne. Les éléments graphiques de base peuvent être stockés dans un tableau alphanumérique et affichés avec l'instruction PUT.

```
DIM dessin$(300)
DIM ecran%(4000,20)
```

Ce système permet de créer un grand écran ne prenant que peu de place. L'écran basse résolution est de 320x200 pixels. Il peut être dessiné avec 20*10 éléments graphiques de base. Si le nombre d'éléments graphiques ne dépasse pas 255, l'écran peut être défini avec des entiers courts 8 bits. La définition d'un écran prend alors 20*10 octets, soit 200 octets. En stockant l'espace virtuel dans un tableau de 4000 lignes, on peut coder 200 écrans pour une taille de 80000 octets (4000 lignes de 20 octets). En comparant la place mémoire prise par 200 écrans avec les 80 000 octets du tableau écran, on arrive à un rapport de 80. Pour faire un calcul exact, il faut tenir compte de la place mémoire prise par les éléments graphiques. Un élément de 16*16 pixels stocké dans une variable alphanumérique prend 134 octets. Les 256 éléments (de 0 à 255) occupent 34304 octets. Le rapport est donc de 200*32000/(80000+34304), soit 56.

```
GET 1,1,16,16,vra$
PRINT LEN(var$)
```

La plupart des jeux d'arcades à scrolling vertical utilisent un système similaire à celui-ci pour créer leurs décors de jeux.

SCROLLING HARD HORIZONTAL SUR STE
Faites danser votre écran

Nous avons vu comment gérer les 4096 couleurs du STE et comment réaliser des scrollings verticaux. Cet article porte sur le scrolling horizontal.

Scrolling horizontal

Au point de vue du spectateur, un scrolling correspond au déplacement d'une fenêtre de visualisation sur une image plus grande que l'écran. Il existe plusieurs techniques de programmation de scrolling. Le système d'affichage du STE peut gérer facilement des scrollings verticaux et horizontaux.
On peut réaliser un scrolling vertical en faisant varier l'adresse de la mémoire vidéo. La fonction `XBIOS(5)` permet de changer l'adresse de la mémoire écran à l'octet près. Etant donné qu'une ligne fait 160 octets en basse résolution, il suffit donc de modifier l'adresse de la mémoire vidéo de 160 octets pour décaler l'écran d'une ligne.

Structure de la mémoire écran STE

La mémoire vidéo basse résolution d'un STF est organisé de manière linéaire. Elle est constituée de 200 lignes de 160 octets pour une taille totale de 32000 octets. Les lignes sont stockées les unes après les autres.

MEMOIRE VIDEO STF 200 LIGNES DE 160 OCTETS	ligne 1 (160 octets) ligne 2 (160 octets) ligne 3 (160 octets) ligne 4 (160 octets)

La mémoire vidéo STE est organisée de la même manière que la mémoire écran STF, c'est-à-dire en ligne de 160 octets. Par contre, il peut y avoir un décalage entre chaque ligne de l'écran.

MEMOIRE VIDEO STE 200 LIGNES DE 160 OCTETS	Zone ligne

Le système d'affichage doit être averti de la valeur du décalage séparant chaque ligne afin de réaliser un affichage correct. Il ne doit lire que les lignes appartenant à l'écran. Ce décalage s'appelle: offset de ligne. Le système d'affichage lit une ligne d'écran en tenantt compte de l'offset de ligne pour calculer l'adresse de la ligne suivante.

La variable système d'adresse `&HFF820F` contient la valeur de l'offset de ligne. Elle est mesurée en mots (1 mot = 2 octets). Si elle contient la valeur 0, il n'y a pas d'espace entre les lignes. La structure de la mémoire vidéo est alors similaire à la mémoire vidéo du STF. C'est le cas le plus courant. L'instruction `SPOKE` permet de modifier le contenu de la variable système `&HFF820F`.

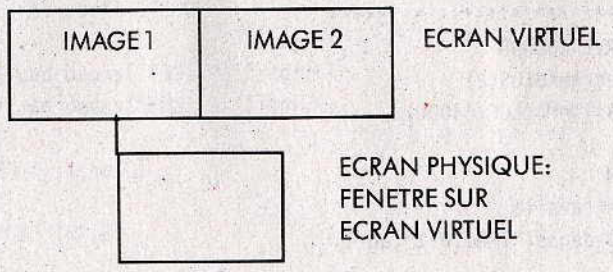
```
SPOKE &HFF820F,0
```

Effet graphique amusant

Lorsque vous modifiez la valeur de l'offset de ligne, la mémoire écran doit avoir une structure tenant compte de cet offset. Si ce n'est pas le cas, l'effet est assez amusant. Avec une valeur différente de 0, le contrôleur vidéo saute des espaces entre chaque ligne. Si l'écran est ordinaire, le contrôleur vidéo affiche sous forme graphique le contenu de la mémoire situé après la zone mémoire vidéo.
Si vous voulez la voir, taper `SPOKE &HFF820F,80`. Le résultat est garanti, surtout avec le bureau GEM. Les barres de l'écran qui bougent sans cesse correspondent aux zones système du ST qui sont continuellement modifiées par le système d'exploitation. Pour annuler cet effet vidéo, taper en aveugle `SPOKE &HFF820F,0`.

Scrolling horizontal

Pour réaliser un scrolling horizontal, on peut déplacer une fenêtre de visualisation sur un écran virtuel. L'écran doit être généré par le programme. La fenêtre de visualisation est l'écran du ST. On modifie le contenu de l'écran en changeant son adresse mémoire grâce aux fonctions spécialisées du STE.



A chaque position de la fenêtre de visualisation correspond une adresse pour la mémoire vidéo. Pour déplacer la position de la fenêtre, il suffit donc de changer une adresse.
Le ST ne tient compte du changement d'adresse vidéo qu'après avoir affiché l'image courante (à la vitesse de cinquante images par secondes).
Si le programme change l'adresse vidéo plus rapidement, seule la dernière modification sera prise en compte.
Pour éviter cela, après chaque changement d'adresse, il faut bloquer l'exécution du programme tant que le ST n'a pas affiché l'image courante. C'est la tâche de l'instruction VSYNC.

```
XBIOS(5)  ! Changement adresse mémoire vidéo
VSYNC     ! Attente fin affichage image courante
```

Puisque le ST affiche cinquante images par seconde, la vitesse de scrolling ne peut pas dépasser cinquante décalages à la seconde. Ceci dit, chaque décalage peut correspondre à autant de pixels que vous le désirez.

Scrolling horizontal 16 pixels par 16 pixels

La structure de l'écran ST en basse résolution est telle que les pixels sont stockés par groupe de 16 dans des blocs de 8 octets. Cette structure particulière est liée au système de codage des pixels par plan de couleurs. On peut réaliser un scrolling horizontal sur un écran virtuel en faisant simultanément varier l'adresse de l'écran et l'offset interligne. Etant donné que le codage interne des pixels se fait par blocs de 16 pixels (8 octets), le scrolling ne peut se faire que par blocs de 16 pixels.
L'écran virtuel fait 64000 octets. L'offset de ligne est de 160 octets, donc de quatre-vingt mots. L'adresse de l'écran est modifiée avec la fonction XBIOS(5). Pour réaliser un scrolling horizontal vers la droite, il faut déplacer l'adresse de l'écran logique de 8 octets à la fois. Chaque déplacement de 8 octets correspond à un déplacement visuel de 16 pixels.

```
adr_pic%=MALLOC(64000)
'
SPOKE &HFF820F,80
FOR bloc%=0 TO 19
 adr%=adr_pic%+bloc%*8
 VOID XBIOS(5,L:-1,L:adr%,-1)
 VSYNC
NEXT bloc%
'
SPOKE &HFF820F,0
VOID MFREE(adr_pic%)
END
```

Ce programme est incomplet. Il s'agit juste d'un exemple montrant la technique de base de scrolling. L'écran virtuel est vide. Il faut le remplir avec une image quelconque pour réaliser une véritable démonstration.
Si vous voulez réaliser un programme complet, inspirez-vous du programme suivant.

Scrolling horizontal pixel par pixel

Pour réaliser un scrolling horizontal pixel par pixel, il faut positionner l'écran physique par rapport à l'écran virtuel à 1 pixel prés. La variable système d'adresse &HFF8265 permet d'y arriver.
Elle contient une valeur que l'on appelle l'*offset de pixel*.
Lorsque le contrôleur vidéo affiche une ligne sur l'écran, il commence par lire le bloc mémoire correspondant aux seize premiers pixels de la ligne. L'*offset de pixel* permet de préciser auquel de ces 16 pixels doit commencer l'affichage. Avec un *offset de pixel* de 0, l'affichage commence au premier pixel du bloc. C'est le mode normal d'affichage (le seul permis par le STF).
Pour réaliser un déplacement de pixel, il suffit de modifier la valeur de la variable &HFF8265. L'exemple suivant réalise un scrolling pixel par pixel sur une longueur de 16 pixels.

```
FOR offset_pixel%=0 TO 15
 VSYNC
 SPOKE &hFF8265,offset_pixel%
NEXT offset_pixel%
```

Si vous utilisez ce programme sur un écran normal, le résultat risque d'être surprenant. Pour obtenir un scrolling correct, il faut créer un espace virtuel dont la largueur est supérieure à celle de l'écran.
L'exemple suivant utilise un écran virtuel de 336 pixels de large. Sa largueur en octets est de 168 octets. L'*offset de ligne* est de 8 octets, donc de quatre mots.
La taille de l'écran virtuel est de 200x168 octets = 33600 octets.

```
' ***************************
' *  SCROLLING HORIZONTAL   *
' *  SUR 16 PIXELS          *
' ***************************
adr_pic%=MALLOC(33600)
'
VOID XBIOS(5,L:-1,L:adr_pic%,-1)
VSYNC
SPOKE &HFF8265,4
'
FOR offset_pixel%=0 TO 15
 VSYNC
 SPOKE &hFF8265,offset_pixel%
NEXT offset_pixel%
'
VOID MFREE(adr_pic%)
END
```

Si vous exécutez ce programme, vous constaterez qu'il ne fonctionne pas. La raison est que l'*offset de ligne* est variable. Lorsque l'*offset de pixel* vaut 0, l'*offset de ligne* est correct. Par contre, lorsque l'*offset de pixel* est différent de 0, le contrôleur vidéo doit lire les pixels restants dans le bloc de 16 pixels suivants. La valeur de l'*offset vidéo* doit alors diminuer de la taille d'un bloc de 16 pixels, c'est-à-dire de 8 octets (quatre mots).
Voici une version corrigée du programme d'exemple. Lorsque l'*offset de pixel* est égal à 0, l'*offset de ligne* vaut 4. Lorsque l'*offset de pixel* est différent de 0, l'*offset de ligne* vaut 0. Le programme de scrolling ho-

rizontal publié dans **Atari Magazine** N°6 donnait un scrolling saccadé car l'*offset de ligne* n'étant pas modifié lorsqu'il était égal à 0.

```
' ************************
' *  SCROLLING HORIZONTAL  *
' *  NON SACCADE           *
' *  SUR 16 PIXELS         *
' ************************
adr_pic%=MALLOC(33600)
'
VOID XBIOS(5,L:-1,L:adr_pic%,-1)
VSYNC
SPOKE &HFF8265,4
'
FOR offset_pixel%=0 TO 15
 VSYNC
 SPOKE &hFF8265,offset_pixel%
 '
 IF offset_pixel%=0
  SPOKE &HFF8265,4
 ELSE
  SPOKE &HFF8265,0
 ENDIF
 '
NEXT offset_pixel%
'
VOID MFREE(adr_pic%)
END
```

Attention: dans ce programme, l'*offset de ligne* vaut 0 si l'*offset de pixel* est différent de 0. C'est un cas particulier qui n'est valable que dans cet exemple simple. Il ne faut pas oublier que si l'*offset de pixel* est différent de 0, l'*offset de ligne* est égal à l'*offset de ligne* théorique moins 4 (dans l'exemple 4-4=0).
Si vous travaillez sur un écran virtuel constitué de deux écrans mis bout à bout (160 octetsx2=320 octets, donc 160 mots), l'*offset de ligne* vaut 80 si l'*offset de ligne* est égal à 0 et 76 (80-4) dans le cas contraire.

Exemple de scrolling horizontal pixel à pixel

Voici un programme de démonstration qui effectue un scrolling horizontal entre deux images *Degas Elite*. Ce scrolling est réalisé en déplaçant une fenêtre sur un écran virtuel. L'écran virtuel est constitué de deux images *Degas Elite* mises bout à bout. Sa taille est de 64000 octets (2x32000 octets).

IMAGE 1	IMAGE 2

Le scrolling est réalisé en déplaçant une fenêtre sur l'écran virtuel. Pour positionner l'écran physique à une position quelconque de l'écran virtuel, il faut calculer l'adresse du bloc de 16 pixels correspondant à cette position, la valeur de l'*offset de pixel* et la valeur de l'*offset de ligne*.

```
offset_pixel%=px% MOD 16
adr_phys%=adr_pic%+(px% DIV 16)*8
IF offset_pixel%=0
 offset_ligne%=80
ELSE
 offset_ligne%=76
ENDIF
```

Le numéro du bloc est obtenu en divisant la position d'affichage par 16. Comme un bloc correspond à 8 octets, on multiplie ensuite le numéro de bloc par 8 afin d'avoir l'adresse du bloc.

```
adr_bloc%=(px% DIV 16)*8
```

Cette adresse est relative à la zone mémoire contenant l'image. Pour obtenir une adresse absolue, il faut y additionner l'adresse de la zone mémoire.

```
adr_bloc%=adr_img%+(px% DIV 16)*8
```

La valeur de l'*offset de pixel* est calculé avec la fonction MODULO.

```
offset_pixel%=px% MOD 16
```

La valeur de l'*offset de ligne* vaut 80 si l'*offset de pixel* est égal à 0 et 76 (80-4) si l'*offset de pixel* est différent de 0.
La procédure `@position_fenetre(px)` calcule automatiquement les paramètres de la fenêtre correspondant à la position px. Une fois les paramètres déterminés, elle positionne l'écran physique à la position correspondant à la position px. Une fois la procédure `@position_fenetre` créée, les routines de scrolling sont particulièrement simples à réaliser. Pour scroller une image vers la gauche, il suffit de positionner la fenêtre sur la position 0 de l'écran virtuel et de déplacer cette position vers la gauche d'un pixel à la fois.

```
@position_fenetre(0)
FOR px%=0 TO 320
 @position_fenetre(px%)
NEXT px%
```

Pour réaliser un scrolling vers la droite, il faut positionner la fenêtre au bout de l'écran virtuel (px=320) et déplacer la fenêtre vers le bord gauche (px=0).

```
@position_fenetre(0)
FOR px%=320 TO 0 STEP -1
 @position_fenetre(px%)
NEXT px%
```

Le programme stocke deux images *Degas Elite* en mémoire et réalise un scrolling allant d'une image à une autre. Les images sont chargées dans la mémoire écran initiale, puis recopiées dans l'écran virtuel en tenant compte de sa structure particulière (160 octets de l'image 1 suivis de 160 octets de l'image 2, etc.)

```
' ************************
' *  SCROLLING HORIZONTAL  *
' ************************
```

```
RESERVE 100000
'
HIDEM
adr_pic%=MALLOC(64000)
'
adr_sauve%=XBIOS(2)
'
' ******************************
' *  CHARGEMENT PREMIERE IMAGE  *
' ******************************
@load_degas("IMAGE1.PI1",XBIOS(2))
adr_ecran%=XBIOS(2)
adr_memoire%=adr_pic%
FOR l%=0 TO 199
  BMOVE adr_ecran%,adr_memoire%,160
  ADD adr_ecran%,160
  ADD adr_memoire%,320
NEXT l%
'
' ******************************
' *  CHARGEMENT SECONDE IMAGE  *
' ******************************
@load_degas("IMAGE2.PI1",XBIOS(2))
adr_ecran%=XBIOS(2)
adr_memoire%=adr_pic%+160
FOR l%=0 TO 199
  BMOVE adr_ecran%,adr_memoire%,160
  ADD adr_ecran%,160
  ADD adr_memoire%,320
NEXT l%
'
CLS
PRINT "Pressez une touche pour"
PRINT "Commencer la démonstration"
VOID INP(2)
'
' ******************************
' *  SCROLLING VERS LA DROITE  *
' ******************************
FOR px%=0 TO 320
  @position_fenetre(px%)
NEXT px%
'
' ******************************
' *  SCROLLING VERS LA GAUCHE  *
' ******************************
FOR px%=320 TO 0
  @position_fenetre(px%)
NEXT px%
'
VOID INP(2)
'
' ************************************
' *  REMISE ECRAN DANS ETAT INITIAL  *
' ************************************
SPOKE &HFF8265,0
SPOKE &HFF820F,0
VOID XBIOS(5,L:-1,L:adr_sauve%,-1)
VSYNC
VOID MFREE(adr_pic%)
SHOWM
END
'
' ************************************
' *  AFFICHAGE DE LA FENETRE ECRAN  *
' ************************************
PROCEDURE position_fenetre(px%)
  LOCAL offset_pixel%
  LOCAL adr_phys%
  LOCAL offset_ligne%
  '
  offset_pixel%=px% MOD 16
  adr_phys%=adr_pic%+(px% DIV 16)*8
  IF offset_pixel%=0
    offset_ligne%=80
  ELSE
    offset_ligne%=76
  ENDIF
  VOID XBIOS(5,L:-1,L:adr_phys%,-1)
  VSYNC
  SPOKE &HFF8265,offset_pixel%
  SPOKE &HFF820F,offset_ligne%
RETURN
'
' ************************************
' *  CHARGEMENT IMAGE DEGAS ELITE  *
' ************************************
PROCEDURE load_degas(nom$,adr%)
  LOCAL pal$
  '
  pal$=SPACE$(32)
  OPEN "i",#1,nom$
  SEEK #1,2
  BGET #1,VARPTR(pal$),32
  VOID XBIOS(6,L:VARPTR(pal$))
  BGET #1,adr%,32000
  CLOSE #1
RETURN
```

La vitesse de scrolling horizontal peut être accélérée en augmentant le pas d'incrémentation entre chaque pixel.
Avec 2 pixels, la vitesse est doublée. Avec 4 pixels, la vitesse est encore doublée.
Remarque: le système de scrolling modifie l'adresse de la mémoire vidéo. Il faut remettre l'écran dans son état initial afin d'éviter les surprises désagréables. Le programme sauve l'adresse initiale dans la variable adr_sauve%.

Scrolling horizontal commandé par la souris

Pour obtenir un effet spectaculaire, vous pouvez asservir le scrolling aux positions de la souris. C'est très facile à réaliser en affichant la fenêtre de visualisation à la position courante de la souris.

```
DO
  xm%=MOUSEX
  EXIT IF MOUSEK<>0
  @position_fenetre(xm%)
LOOP
```

Cette routine fonctionne bien, mais elle présente un défaut. La fenêtre de visualisation est sans cesse redessinée à l'écran. Pour éviter cela, il faut écrire une routine qui attende un déplacement de la souris pour redessiner la fenêtre de visualisation.

```
x0%=MOUSEX
@position_fenetre(x0%)
DO
  DO
    xm%=MOUSEX
    EXIT IF xm%<>x0%
    EXIT IF MOUSEK<>0
  LOOP
  EXIT IF MOUSEK<>0
  @position_fenetre(xm%)
  x0%=xm%
LOOP
```

La position de départ de la souris est mémorisée dans la variable x0%. Une boucle DO-LOOP bloque l'exécution du programme tant que la souris ne change pas de position x. Etant donné qu'il y a deux boucles DO-LOOP, la condition de sortie (clic sur la souris) est testée deux fois (EXIT IF km%<>0).
Pour obtenir le programme complet, il faut reprendre le programme de scrolling et le modifier légèrement.
Remarque: dans le programme précédent, la fenêtre de visualisation est déplacée de la position 0 à la position 320. Etant donné que la position de la souris ne peut varier qu'entre 0 et 319, la dernière colonne de l'écran virtuel correspondant à la position 320 ne sera jamais affichée à l'écran.

Utilisation simultanée des scrollings verticaux et horizontaux

Après le scrolling vertical et le scrolling horizontal, voici maintenant un programme d'exemple qui utilise simultanément les deux modes de scrolling. Il affiche une fenêtre sur un écran virtuel de 640x400 pixels. Cet écran virtuel est remplis avec quatre images *Degas Elite*. La fenêtre de visualisation est déplacée avec la souris. L'écran virtuel occupe quatre images *Degas Elite*, donc 128000 octets (4x32000 octets). Les lignes d'une même image sont séparées par 160 octets. Pour calculer l'adresse de la ligne n+1 à partir de l'adresse de la ligne n, il faut lui ajouter la valeur 320.
Ce chiffre correspond à la longueur de la ligne (160 octets), additionné à la longueur de l'espace interligne (160 octets). On obtient donc un total de 320 octets.
L'image IMAGE 1 commence à l'adresse de l'écran virtuel. L'image IMAGE 2 commence à l'adresse de l'écran virtuel plus 160 octets. L'image IMAGE 3 commence à l'adresse de l'écran virtuel plus 32000 octets. Et enfin, l'image IMAGE 4 commence à l'adresse de l'écran virtuel plus 32160 octets (32000+160).

IMAGE 1	IMAGE 2
IMAGE 3	IMAGE 4

La procédure @fixe_ecran(px,py) positionne la fenêtre de visualisation à la position (px,py). Nous avons vu plus haut que la position de la mémoire écran est définie par une adresse, un *offset de ligne* et un *offset de pixel*. Les lignes sont séparées par 80 mots (160 octets). L'*offset de ligne* est donc de 80 lorsque l'*offset de pixel* est égal à 0, et de 76 (80-4) dans le cas inverse. L'*offset de pixel* est défini avec l'opérateur MODULO.

```
offset_pixel%=px% MOD 16
```

L'adresse de l'écran est calculée à partir de la position x et de la position y. La position y sert à déterminer l'adresse de la ligne. La position x sert à calculer l'adresse relative du début de l'écran par rapport à l'adresse de la ligne.

```
adr_ligne%=adr_pic%+py%*320
adr_phys%=adr_ligne%+(px% DIV 16)*8
```

Le programme commence par remplir l'écran virtuel avec les quatre images *Degas Elite*. Les images sont chargées avec la procédure @load_degas dans une mémoire tampon, puis recopiées à leurs places respectives dans l'écran virtuel. L'utilisation d'une mémoire tampon permet d'éviter l'affichage des images à l'écran lors de l'initialisation du programme.

```
' ************************************
' *  SCROLLING SUR UN ECRAN VIRTUEL  *
' *  DE 640*400 (4 IMAGES DEGAS)     *
' ************************************
' * (C) ATARI MAGAZINE   1991        *
' * (C) PATRICK LECLERCQ 1991        *
' ************************************
' *  PROGRAMMATION GFA BASIC 3.xx    *
' ************************************
RESERVE 100000
'
HIDEM
adr_tampon%=MALLOC(32000)
adr_pic%=MALLOC(128000)
'
adr_sauve%=XBIOS(2)
'
' ******************************
' *  CHARGEMENT PREMIERE IMAGE  *
' ******************************
@load_degas("IMAGE1.PI1",adr_tampon%)
adr_ligne%=adr_tampon%
```

```
adr_copie%=adr_pic%
FOR l%=1 TO 200
  BMOVE adr_ligne%,adr_copie%,160
  ADD adr_ligne%,160
  ADD adr_copie%,320
NEXT l%
'
' ******************************
' *  CHARGEMENT SECONDE IMAGE  *
' ******************************
@load_degas("IMAGE2.PI1",adr_tampon%)
adr_ligne%=adr_tampon%
adr_copie%=adr_pic%+160
FOR l%=1 TO 200
  BMOVE adr_ligne%,adr_copie%,160
  ADD adr_ligne%,160
  ADD adr_copie%,320
NEXT l%
'
' *******************************
' *  CHARGEMENT TROISIEME IMAGE  *
' *******************************
@load_degas("IMAGE3.PI1",adr_tampon%)
adr_ligne%=adr_tampon%
adr_copie%=adr_pic%+64000
FOR l%=1 TO 200
  BMOVE adr_ligne%,adr_copie%,160
  ADD adr_ligne%,160
  ADD adr_copie%,320
NEXT l%
'
' *******************************
' *  CHARGEMENT QUATRIEME IMAGE  *
' *******************************
@load_degas("IMAGE4.PI1",adr_tampon%)
adr_ligne%=adr_tampon%
adr_copie%=adr_pic%+64000+160
FOR l%=1 TO 200
  BMOVE adr_ligne%,adr_copie%,160
  ADD adr_ligne%,160
  ADD adr_copie%,320
NEXT l%
'
' ******************************
' *    INITIALISATION DE LA    *
' *  FENETRE DE VISUALISATION  *
' ******************************
SETMOUSE 160,100
MOUSE xm%,ym%,km%
@fixe_ecran(xm%,ym%)
x0%=xm%
'
' ************************
' *  BOUCLE PRINCIPALE  *
' ************************
DO
  DO
    MOUSE xm%,ym%,km%
    EXIT IF (xm%<>x0%) OR (ym%<>y0%)
    EXIT IF km%<>0
  LOOP
  EXIT IF km%<>0
  @fixe_ecran(xm%,ym%)
  x0%=xm%
  y0%=ym%
LOOP
'
' *********************************
' *  REINITIALISATION DE L'ECRAN  *
' *********************************
SPOKE &HFF8265,0
SPOKE &HFF820F,0
VOID XBIOS(5,L:-1,L:adr_sauve%,-1)
VSYNC
VOID MFREE(adr_pic%)
VOID MFREE(adr_tampon%)
SHOWM
END
'
' ****************************
' *  MODIFICATION POSITION  *
' *      ECRAN VIRTUEL      *
' ****************************
PROCEDURE fixe_ecran(px%,py%)
  LOCAL offset_pixel%
  LOCAL adr_phys%
  LOCAL offset_ligne%
  LOCAL adr_ligne%
  '
  adr_ligne%=adr_pic%+py%*320
  offset_pixel%=px% MOD 16
  adr_phys%=adr_ligne%+(px% DIV 16)*8
  IF offset_pixel%=0
    offset_ligne%=80
  ELSE
    offset_ligne%=76
  ENDIF
  VOID XBIOS(5,L:-1,L:adr_phys%,-1)
  VSYNC
  SPOKE &HFF8265,offset_pixel%
  SPOKE &HFF820F,offset_ligne%
RETURN
'
' **********************
' *  CHARGEMENT IMAGE  *
' *    DEGAS ELITE     *
' **********************
PROCEDURE load_degas(nom$,adr%)
  LOCAL pal$
  '
  pal$=SPACE$(32)
  OPEN "i",#1,nom$
  SEEK #1,2
  BGET #1,VARPTR(pal$),32
```

```
  VOID XBIOS(6,L:VARPTR(pal$))
  BGET #1,adr%,32000
  CLOSE #1
RETURN
```

L'utilisation d'une mémoire tampon pour le chargement des images permet d'éviter leur affichage à l'écran. Le tampon prend 32000 octets de mémoire. Pour éviter cela, on peut écrire une routine qui charge directement une image *Degas Elite* dans l'écran tampon.

```
PROCEDURE charge_degas(nom$,adr_image%)
  LOCAL pal$
  LOCAL ligne%,adr_ligne%
  '
  pal$=SPACE$(32)
  OPEN "i",#1,nom$
  SEEK #1,2
  BGET #1,VARPTR(pal$),32
  VOID XBIOS(6,L:VARPTR(pal$))
  '
  adr_ligne%=adr_image%
  FOR ligne%=1 TO 200
    BGET #1,adr_ligne%,160
    ADD adr_ligne%,320
  NEXT ligne%
  CLOSE #1
NEXT ligne%
```

Le chargement des images dans l'écran virtuel est maintenant beaucoup plus facile à programmer.

```
@charge_degas("IMAGE1.PI1",adr_pic%)
@charge_degas("IMAGE2.PI1",adr_pic%+160)
@charge_degas("IMAGE3.PI1",adr_pic%+32000)
@charge_degas("IMAGE4.PI1",adr_pic%+32160)
```

Attention: la routine `@charge_degas` est une routine spécifique qui ne fonctionne qu'avec un écran virtuel d'une largeur de 640 pixels (320 octets de large).
Pour avoir une routine fonctionnant avec un autre écran virtuel, il faut modifier la ligne `ADD adr_ligne%,320` en tenant compte de la largueur du nouvel écran.

Déplacement souris sur un écran virtuel de taille quelconque

Les déplacements de la fenêtre de visualisation étant liés aux déplacements de la souris, la dernière ligne et la dernière colonne de l'écran virtuel ne sont jamais affichés par le programme. On ne peut utiliser le système précédent que pour un écran de 639x399 pixels. Pour déplacer la fenêtre de visualisation avec la souris sur un écran virtuel de taille quelconque (mais supérieur à l'écran visible), il faut gérer la souris de manière différente.
La position X de la souris varie entre 0 et 319, et la position Y varie entre 0 et 199 pixels. Pour obtenir un déplacement sur un écran de dimension quelconque, il ne faut pas tenir compte des déplacements absolus de la souris, mais des déplacements relatifs. Pour obtenir un déplacement relatif, il faut positionner la souris à un emplacement précis de l'écran et attendre que sa position change. La valeur du déplacement relatif est la différence entre la nouvelle position de la souris et sa position initiale. Dès que le déplacement relatif est calculé, le programme doit repositionner la souris sur sa position initiale.
Après un déplacement souris, le programme doit calculer la nouvelle position de la fenêtre de visualisation. Ce calcul est effectué en additionnant le déplacement relatif avec la position courante de la fenêtre de visualisation. Il peut arriver que le calcul du déplacement donne des coordonnées en dehors de l'écran. Pour éviter cela, le programme doit exercer un contrôle sur les coordonnées de la fenêtre. Les variables `xmax%` et `ymax%` contiennent les positions extrêmes de la fenêtre. Les positions minimales sont (0,0).

```
xmax%=320
ymax%=200
'
SETMOUSE 160,100
xaff%=100
yaff%=100
@fixe_ecran(xaff%,yaff%)
'
DO
  ' *************************
  ' * ATTENTE ACTION SOURIS *
  ' *************************
  DO
    MOUSE xm%,ym%,km%
    EXIT IF (xm%<>160) OR (ym%<>100)
    EXIT IF km%<>0
  LOOP
  EXIT IF km%<>0
  '
  ' ******************************
  ' * CALCUL POSITION X FENETRE *
  ' ******************************
  IF xm%<>160
    dx%=xm%-160
    ADD xaff%,dx%
    IF xaff%<0
      xaff%=0
    ENDIF
    IF xaff%>xmax%
      xaff%=xmax%
    ENDIF
  ENDIF
  '
  ' ******************************
  ' * CALCUL POSITION Y FENETRE *
  ' ******************************
  IF ym%<>100
    dy%=ym%-100
    ADD yaff%,dy%
    IF yaff%<0
      yaff%=0
    ENDIF
    IF yaff%>ymax%
```

```
      yaff%=ymax%
    ENDIF
  ENDIF
  '
  SETMOUSE 160,100
  @fixe_ecran(xaff%,yaff%)
LOOP
```

Remarque: en modifiant les valeurs de position extrêmes de la souris, on peut limiter le scrolling à une zone précise de l'écran virtuel. Cela peut être utile dans certaines applications. L'exemple suivant limite les déplacements de la fenêtre de visualisation entre les coordonnées (50,50) et (200,150).

```
xmin%=50
ymin%=50
xmax%=200
ymax%=150
'
.....
'
IF xaff%<xmin%
  xaff%=xmin%
ENDIF
IF xaff%>xmax%
  xaff%=xmax%
ENDIF
'
IF yaff%<ymin%
  yaff%=ymin%
ENDIF
IF yaff%>ymax%
  yaff%=ymax%
ENDIF
```

Taille des écrans virtuels

La place mémoire prise par les écrans virtuels croit rapidement en fonction de leurs tailles. Un écran virtuel de 2x2 images prend 128 Ko. Un écran de 3x3 images prend 288 Ko. La taille d'un écran virtuel doit correspondre à certaines conditions: elle doit être supérieure ou égal à la taille d'un écran normal et la largeur de l'écran doit être un multiple de 16.
Par exemple, on peut créer un écran virtuel de 592x257 pixels. Une largeur de 592 pixels correspond à 37 blocs de 16 pixels. Les lignes de cet écran ont une taille de 296 octets (un bloc de 16 pixels = 8 octets). La taille mémoire de l'écran est de 296x257 octets, soit 76072 octets.

Pour finir

Maintenant que vous savez gérer les 4096 couleurs et le scrolling hard du STE, vous pouvez réaliser des programmes intéressants et de superbes démos. Envoyez-nous vos réalisations.

OFFRE SPÉCIALE

Pour rester branché ST, TT, FALCON et le reste Abonnez-vous à

Start Micro Magazine

Chaque mois:
- des infos,
- des tests,
- des conseils,
- des initiations,

+

1 disquette pleine de programmes, d'images, de sons, etc.

Votre abonnement vous donne accès à "3615 MICROHELP"
Votre abonnement vous procurera des réductions intéressantes sur le téléchargement et l'achat de disquettes de logiciels du domaine public
Votre abonnement vous fera bénéficier de tarifs promotionnels sur certains logiciels du commerce

Abonnez-vous et économisez immédiatement

80 F

Bulletin d'abonnement page 130

EFFETS SPECIAUX GRAPHIQUES
Faites flasher vos écrans

Les effets vidéo permettent d'agrémenter vos applications graphiques et vos jeux. Nous vous proposons quelques conseils et explications pour bien en comprendre le fonctionnement.

Instruction BMOVE

Elle permet de copier très rapidement une zone mémoire (environ 1600 ko à la seconde). On l'utilise surtout pour les applications graphiques.

```
BMOVE adr_org, adr_dest, nb_octets,
adr_org  : adresse de la zone mémoire d'origine,
adr_dest : adresse de la zone mémoire de destination,
nb_octets : nombre d'octets à copier.
```

Structure mémoire de l'écran ST

L'écran du ST a une taille de 32000 octets. Le codage des informations graphiques est différent selon le mode graphique utilisé.
La basse résolution est organisée en 200 lignes de 320 pixels avec 16 couleurs possibles par pixel. Chaque ligne a une taille de 160 octets. Un octet contient donc la représentation binaire de 2 pixels. C'est le mode graphique le plus utilisé pour les jeux.
Les programmes donnés en exemple dans cet article sont prévus pour la basse résolution.
La moyenne résolution est organisée en 200 lignes de 640 pixels avec 4 couleurs par pixel. Les lignes font toujours 160 pixels, mais un octet contient la représentation binaire de 4 pixels. C'est un mode graphique rarement utilisé pour les jeux. Il sert pour les applications professionnelles réalisées par les utilisateurs qui n'ont pas de moniteur monochrome haute résolution.
La haute résolution fait 400 lignes de 640 pixels en deux couleurs. Les lignes ont une taille de 80 octets. Un octet contient la représentation binaire de 8 pixels. Ce mode graphique nécessite un moniteur monochrome de haute performance. La plupart des applications professionnelles utilisent la haute résolution.

Manipulation de l'écran avec BMOVE

L'instruction BMOVE permet de copier des blocs de mémoire et donc des parties de l'écran. On peut copier le contenu de l'écran en mémoire et ensuite le réafficher.

```
' ************************************
' * SAUVEGARDE DE L'ECRAN EN MEMOIRE *
' ************************************
adr_tampon% = MALLOC(32000)        !RESERVATION MEMOIRE
BMOVE XBIOS(2),adr_tampon%,32000 !COPIE ECRAN MEMOIRE
CLS                                !EFFACEMENT ECRAN
VOID INP(2)                        !ATTENTE CLAVIER
BMOVE adr_tampon%,XBIOS(2),32000 !AFFICHAGE ECRAN
VOID INP(2)                        !ATTENTE CLAVIER
VOID MFREE(adr_tampon%)            !LIBERATION MEMOIRE
```

Cette technique fonctionne dans toutes les résolutions du ST. La fonction «File select» et les boîtes d'alerte utilisent cette méthode (en fait, elles ne sauvent pas toute l'image, mais seulement la partie modifiée).

Manipulation des lignes

En faisant des manipulations graphiques des lignes, on obtient des effets intéressants. Il faut connaître l'adresse des lignes pour pouvoir les copier grâce à BMOVE. Le numéro des lignes varie entre 0 et 199 pour la basse et la moyenne résolution, et entre 0 et 399 pour la haute résolution.

Adresse d'une ligne: `Adr_ligne = Adr_ecran+Ligne*Taille_ligne`
Basse résolution: `Adr_ligne = XBIOS(2)+ligne*160`
Moyenne résolution: `Adr_ligne = XBIOS(2)+ligne*160`
Haute résolution: `Adr_ligne = XBIOS(2)+ligne*80`

Exemple simple

On peut faire un petit programme qui copie la première ligne d'un écran sur tout l'écran.

```
PROCEDURE copie_ligne LOCAL adr_ligne0%
LOCAL adr_copie%
LOCAL ligne_copie%
' --
```

```
adr_ligne0% = XBIOS(2)              !ADR LIGNE A COPIER
FOR ligne_copie% = 1 TO 199
  adr_copie% = XBIOS(2)+ligne_copie%*160  !ADR LIGNE
  BMOVE adr_ligne0%, adr_copie%, 160      !COPIE
  NEXT ligne_copie%
RETURN
```

Amélioration de l'exemple

Au lieu de copier une seule ligne, on peut en copier plusieurs à la fois. Pour cet exemple, on divise l'écran en 20 bandes de 10 lignes. Elles sont numérotées de 0 à 19. La routine copie la bande 0 sur les autres bandes.

```
PROCEDURE copie_bande
LOCAL adr_org%
LOCAL adr_copie%
' ---
! ADRESSE BANDE 0
adr_org% = XBIOS(2)
FOR bande_copie% = 1 TO 19
  ' CALCUL ADRESSE BANDE 2
  adr_copie% = XBIOS(2)+10*bande_copie%*160
  ' COPIE BANDE 1 SUR BANDE 2
  BMOVE adr_org%, adr_copie%, 1600
NEXT bande_copie%
RETURN
```

Utilisation de plusieurs écrans en mémoire

Ces petits programmes sont amusants, mais peu utiles. En revanche, l'intérêt est de réaliser des affichages à partir d'un ou plusieurs écrans stockés en mémoire.
La place mémoire est reservée avec l'instruction MALLOC. La routine load_degas permet de charger une image *Degas Elite* à n'importe quelle adresse en mémoire. On peut réaliser toutes sortes d'effets comme l'affichage rapide d'une image, l'affichage à partir du haut ou du bas de l'écran, l'inversion d'une image, l'affichage à partir du centre ou des bords, etc. Il faut tenir compte de l'adresse des écrans dans le calcul des adresses des lignes. L'adresse de base de l'écran physique (celui qui est visible sur le moniteur de l'ordinateur), est fournie par la fonction XBIOS(2). Il ne faut surtout pas se tromper au niveau des adresses de base des écrans.

Affichage rapide d'une image

La technique la plus simple pour afficher rapidement une image est de la charger dans une mémoire tampon, puis de la copier sur l'écran avec BMOVE. Etant donné la vitesse de BMOVE, l'affichage est instantané pour l'utilisateur (en fait, il prend à peu près 20 millisecondes). Avec une routine de chargement directement en mémoire, l'affichage est lent et saccadé.

Affichage à partir du bas de l'écran

Pour afficher une image à partir du bas de l'écran, il suffit de la charger dans une mémoire tampon, puis d'afficher les lignes les unes après les autres de la ligne 199 à la ligne 0. La routine aff_bas du programme d'exemple montre que cette méthode est très simple à programmer.

Affichage à partir du haut de l'écran

Pour afficher une image à partir du haut de l'écran, il suffit de la charger dans une mémoire tampon, puis d'afficher les lignes les unes après les autres de la ligne 0 à la ligne 199. La routine aff_haut du programme d'exemple fonctionne sur ce principe.

Affichage à partir du centre de l'écran

Pour afficher une image stockée dans la mémoire sur l'écran à partir du centre, il faut afficher d'abord les lignes 99 et 100, puis les lignes 98 et 101, et ainsi de suite jusqu'aux lignes 0 et 199. L'effet est amusant. La routine aff_milieu du programme en exemple utilise cette technique.

Affichage à partir des bords de l'écran

Cette méthode est l'inverse de la précédente. Il faut d'abord afficher les lignes 0 et 199, puis 1 et 198, pour arriver finalement aux lignes 99 et 100. La routine aff_bord illustre cette technique.

Affichage aléatoire

Cette méthode consiste à afficher les lignes d'une image dans un ordre aléatoire. Le numéro de la ligne à afficher est déterminé par une fonction aléatoire. Il faut tenir compte des lignes déjà affichées pour éviter de les réafficher. La solution la plus simple est d'utiliser un tableau ne contenant que des 0. Chaque fois qu'une ligne est affichée, on met l'indice du tableau correspondant à 1. Pour chercher une ligne non affichée, il suffit de lire le tableau à la recherche d'un nombre 0.
La routine aff_aléatoire du programme d'exemple montre une réalisation de cette technique. Cette routine utilise un tableau tst_ligne% pour la gestion des lignes. Elle fait un affichage ligne à ligne, mais on peut la modifier afin de travailler sur des blocs de lignes. L'affichage de l'image se fera plus vite puisqu'il y aura moins d'étapes d'affichage. Le tableau tst_ligne prendra aussi moins de place (économie de mémoire).

Inversion verticale d'une image

Pour inverser verticalement une image, il suffit de la recopier en inversant la position des lignes. La ligne 0 devient la ligne 199, la ligne 1 devient la ligne 198, la ligne 2 devient la ligne 197, .., la ligne 199 devient la ligne 0.
La routine inversion_ecran inverse l'image de l'écran physique et stocke le résultat dans une image tampon. Une fois l'inversion terminée, elle donne le résultat en le recopiant rapidement sur l'écran physique.

Scrolling vertical d'une image

Pour faire scroller une image sur une zone d'écran de hauteur h, il faut copier le bloc de lignes de hauteur h à partir de la ligne 0 de l'i-

mage, puis à partir de la ligne 1 de l'image, etc. La superposition rapide des blocs donnera l'impression d'un mouvement.
En affichant l'image en bas de l'écran, on arrive à faire scroller un bloc de 100 lignes de haut d'une manière parfaite sans la moindre «bavure» vidéo. La routine scroll du programme d'exemple fait d'abord un scrolling vers le bas, puis vers le haut.

Programmes d'exemples

Les routines d'affichages ont été réalisées en *GFA Basic 3.03*. Les exemples utilisent deux images *Degas Elite* chargées en mémoire au début du programme. Le choix des images se fait grâce au sélecteur de fichiers.

Problèmes de synchronisation vidéo et instruction VSYNC

L'image affichée par un moniteur vidéo n'est pas une image fixe. En fait, c'est un spot vidéo unique qui se déplace à grande vitesse pour dessiner l'image. Ce spot balaie l'écran 50 fois par seconde (en terme technique, on dit que l'écran est rafraîchi à 50 Hz). Donc, 50 fois par seconde, le système d'affichage parcourt la mémoire vidéo du début à la fin (soit 32000 octets). Entre chaque balayage vidéo, il y a un petit temps mort (le temps de balayage est de 16 millisecondes, et le temps entre deux balayages de 4 millisecondes).
Lors de l'affichage d'un objet sur l'écran, celui-ci se met parfois à clignoter, l'objet peut disparaître en partie, une barre horizontale peut apparaître, l'affichage peut être légèrement brouillé, etc. Ces effets parasites viennent d'une interférence entre le système d'affichage et la routine graphique. Si le programme modifie une donnée de la mémoire écran au moment où le système d'affichage envoie cette donnée sur l'écran, il se produit une perturbation graphique.
Pour éviter ce problème, il faut travailler pendant le temps libre entre chaque balayage vidéo. La fonction Vsync arrête le déroulement du programme pendant le balayage vidéo. Après l'instruction Vsync, le programme dispose d'un temps très court pour faire un affichage graphique sans craindre de problème vidéo.

Travail en bas de l'écran

Le spot se déplace sur l'écran à une vitesse bien définie. Après un Vsync, on dispose de peu de temps pour faire un affichage sans parasite vidéo en haut de l'écran. En revanche, le temps disponible pour afficher quelque chose vers le bas de l'écran est plus important, puisqu'on dispose du temps entre deux balayages vidéo, plus le temps pris par le système d'affichage pour arriver en bas de l'écran.
La routine de scrolling travaille en bas de l'écran pour profiter de ce temps supplémentaire. Remontez la position d'affichage, et vous verrez apparaître une barre de synchronisation.
La vitesse de rafraîchissement vidéo est de 50 Hz pour tous les moniteurs couleur et de 70 Hz pour le moniteur monochrome Atari.

POUR TOUTES QUESTIONS, TAPEZ
3615 ATARI

```
' ******************************************
' *                                        *
' *  EFFETS GRAPHIQUES AVEC FONCTION BMOVE *
' *  .                                     *
' ******************************************
' ************************************************
' *  (C) 1990 Patrick Leclercq & ATARI MAGAZINE  *
' ************************************************
' *  Programmation en GFA BASIC 3.03  *
' ************************************
'
RESERVE 50000          ! RESERVATION MEMOIRE POUR BASIC
'
DIM tst_ligne%(200)    !TABLEAU POUR AFFICHAGE ALEATOIRE
'
image1%=MALLOC(32000)        ! RESERVATION MEMOIRE
image2%=MALLOC(32000)        ! RESERVATION MEMOIRE
image_travail%=MALLOC(32000)! RESERVATION MEMOIRE
@main
VOID
MFREE(image_travail%)        ! LIBERATION MEMOIRE
VOID MFREE(image2%)          ! LIBERATION MEMOIRE
VOID MFREE(image1%)          ! LIBERATION MEMOIRE
END                          ! FIN DU PROGRAMME
'
' ____________________________________________
' ************************************
' *  ATTENTE D'UN EVENEMENT CLAVIER  *
' ************************************
PROCEDURE att_clavier
  VOID INP(2)
RETURN
'
' ____________________________________________
' ***************************************************
' *  CHARGEMENT D'UNE IMAGE DEGAS ELITE EN MEMOIRE  *
' ***************************************************
PROCEDURE load_degas(nom$,adr%)
  LOCAL pal$
  '
  pal$=SPACE$(32)
  OPEN "i",#99,nom$
  SEEK #99,2
  BGET #99,VARPTR(pal$),32
  VOID XBIOS(6,L:VARPTR(pal$))
  BGET #99,adr%,32000
  CLOSE #99
RETURN
'
'____________________________________________
' ***********************************************
' *  COPIE D'UNE LIGNE D'UN ECRAN SUR UN AUTRE  *
' ***********************************************
PROCEDURE copie_ligne(l1%,adr_img1%,l2%,adr_img2%)
  LOCAL adr1%
  LOCAL adr2%
  '
```

```
  adr1%=adr_image1%+l1%*160 ! ADRESSE DE LA LIGNE 1
  adr2%=adr_image2%+l2%*160 ! ADRESSE DE LA LIGNE 2
  '
  BMOVE adr1%,adr2%,160     ! COPIE LIGNE 1 SUR LIGNE 2
RETURN
'
'_____________________________________________
' ************************************************
' *  AFFICHAGE D'UN BLOCK DE LIGNES SUR L'ECRAN  *
' ************************************************
PROCEDURE aff_block(l1%,nl%,adr_img1%,l2%,adr_img2%)
  LOCAL adr1%
  LOCAL adr2%
  LOCAL taille%
  '
  adr1%=l1%*160+adr_img1%    ! ADRESSE BLOC 1
  adr2%=l2%*160+adr_img2%    ! ADRESSE BLOC 2
  taille%=nbl%*160           ! TAILLE DU BLOC
  '
  VSYNC                      ! ATTENTE SYNCHRO VIDEO
  BMOVE adr1%,adr2%,taille%  ! COPIE DU BLOC
RETURN
'
'_____________________________________________
' ******************************************************
' *  AFFICHAGE DE L'IMAGE A PARTIR DU BAS DE L'ECRAN  *
' ******************************************************
PROCEDURE aff_bas(adr_image%)
  LOCAL ligne%
  '
  FOR ligne%=199 TO 0 STEP -1
    VSYNC
    @copie_ligne(ligne%,adr_image%,ligne%,XBIOS(2))
  NEXT ligne%
RETURN
'
'_____________________________________________
' ******************************************************
' *  AFFICHAGE DE L'IMAGE A PARTIR DU HAUT DE L'ECRAN *
' ******************************************************
PROCEDURE aff_haut(adr_image%)
  LOCAL ligne%
  '
  FOR ligne%=0 TO 199
    VSYNC
    @copie_ligne(ligne%,adr_image%,ligne%,XBIOS(2))
  NEXT ligne%
  '
RETURN
'
'_____________________________________________
' ************************************************
' * AFFICHAGE IMAGE A PARTIR DU MILIEU DE L'ECRAN  *
' ************************************************
PROCEDURE aff_milieu(adr_image%)
  LOCAL ligne1%
  LOCAL ligne2%
  LOCAL i%
  '
  ligne1%=99
  ligne2%=100
  '
  FOR i%=1 TO 100
    VSYNC
    @copie_ligne(ligne1%,adr_image%,ligne1%,XBIOS(2))
    @copie_ligne(ligne2%,adr_image%,ligne2%,XBIOS(2))
    DEC ligne1%
    INC ligne2%
  NEXT i%
RETURN
'
'_____________________________________________
' *********************
' *  AFFICHAGE IMAGE  *
' *********************
PROCEDURE aff_bords(adr_image%)
  LOCAL ligne1%
  LOCAL ligne2%
  LOCAL i%
  '
  ligne1%=0
  ligne2%=199
  '
  FOR i%=1 TO 100
    VSYNC
    @copie_ligne(ligne1%,adr_image%,ligne1%,XBIOS(2))
    @copie_ligne(ligne2%,adr_image%,ligne2%,XBIOS(2))
    INC ligne1%
    DEC ligne2%
  NEXT i%
RETURN
'
'_____________________________________________
' ************************************
' *  AFFICHAGE ALEATOIRE D'UNE IMAGE  *
' ************************************
PROCEDURE aff_aleatoire(adr_image%)
  LOCAL ligne%
  LOCAL i%
  '
  FOR i%=0 TO 199  ! INIT TABLEAU DE LIGNES AFFICHES
    tst_ligne%(i%)=0
  NEXT i%
  '
  FOR i%=1 TO 200
    REPEAT     ! RECHERCHE D'UNE LIGNE NON AFFICHE
      ligne%=RANDOM(200)
    UNTIL tst_ligne%(ligne%)=0
    tst_ligne%(ligne%)=1          ! MARQUAGE LIGNE
    @copie_ligne(ligne%,adr_image%,ligne%,XBIOS(2))
  NEXT i%
RETURN
'
'_____________________________
```

```
' ************************************
' *  INVERSION COMPLETE DE L'ECRAN  *
' ************************************
PROCEDURE inversion_ecran
  LOCAL ligne1%
  LOCAL ligne2%
  '
  ligne2%=199
  '
  FOR ligne1%=0 TO 199
@copie_ligne(ligne1%,XBIOS(2),ligne2%,image_travail%)
    DEC ligne2%
  NEXT ligne1%
  '
  VSYNC
  BMOVE image_travail%,XBIOS(2),32000
RETURN
'
' ___________________________
' ***************************
' *  ROUTINE DE SCROLLING  *
' ***************************
PROCEDURE scroll(adr_image%)
  LOCAL ligne%
  '
  FOR ligne%=0 TO 100 STEP 1
    aff_block(ligne%,100,adr_image%,100,XBIOS(2))
  NEXT ligne%
  '
  FOR ligne%=100 TO 0 STEP -1
    @aff_block(ligne%,100,adr_image%,100,XBIOS(2))
  NEXT ligne%
RETURN
'
' ___________________________________
' ***********************************
' *  DEMONSTRATION DU SCROLLING  *
' ***********************************
PROCEDURE demo_scroll(adr_image%)
  LOCAL i%
  '
  FOR i%=1 TO 2
    @scroll(adr_image%)
  NEXT i%
RETURN
'
' ____________________
PROCEDURE demo_graph
  '
  @aff_haut(image1%)
  @att_clavier
  @aff_haut(image2%)
  @att_clavier
  '
  @aff_bas(image1%)
  @att_clavier
  @aff_bas(image2%)
  @att_clavier
  '
  @aff_bords(image1%)
  @att_clavier
  @aff_bords(image2%)
  @att_clavier
  '
  @aff_milieu(image1%)
  @att_clavier
  @aff_milieu(image2%)
  @att_clavier
  '
  @inversion_ecran
  @att_clavier
  @inversion_ecran
  @att_clavier
  '
  @aff_aleatoire(image1%)
  @att_clavier
  @aff_aleatoire(image2%)
  @att_clavier
  '
  @scroll(image1%)
  @att_clavier
  @scroll(image2%)
  @att_clavier
  '
RETURN
'
' ________________________
' ************************
' *  ROUTINE PRINCIPALE  *
' ************************
PROCEDURE main
  LOCAL fond$
  LOCAL f2$
  '
  FILESELECT "*.PI1"," ",fond$ ! SAISIE NOM IMAGE
  @load_degas(fond$,image1%)   ! CHARGEMENT IMAGE
  '
  FILESELECT "*.PI1"," ",f2$
  @load_degas(f2$,image2%)
  '
  @att_clavier
  @demo_graph
  '
RETURN
```

SI VOUS N'AVEZ PAS LE COURAGE DE TAPER TOUS LES LISTINGS, COMMANDEZ LA DISQUETTE BEST OF GFA BASIC AVEC LES IMAGES AU PRIX DE 70 F SEULEMENT.

Bon de commande en page 50

4096 IMAGES SUR UNE DISQUETTE

Economisez de la place mémoire

Les images des jeux d'arcades sont construites à partir d'élements graphiques collés les uns contre les autres, codant ainsi une image écran en quelques centaines d'octets.

Les éléments dépendent du type de décor. Pour dessiner un temple (sujet classique des jeux d'arcades/aventures), il faut des colonnes, des murs, des symboles bizarres, des portes, des éléments de décors, etc. Les colonnes peuvent être conçues de plusieurs éléments: un socle de base, un élément intermédiaire et un élément terminal. Pour dessiner une colonne à un endroit quelconque, il faut d'abord mettre le socle, puis un ou deux éléments intermédiaires avant de mettre l'élément terminal. La taille de la colonne n'est déterminée que par le nombre d'éléments intermédiaires. Avec les mêmes éléments de base, on peut avoir de grandes ou de toutes petites colonnes.

Pour dessiner les murs, trois ou quatre éléments différents suffisent. En les alternant, on peut créer un mur tout à fait valable. Pour rompre la monotonie, on peut aussi insérer des éléments de décors comme des vases, des autels ou des dessins étranges reproduisant les traits d'une divinité quelconque.

On peut dessiner un temple avec environ 40 éléments graphiques de base. Ils peuvent être utilisés pour dessiner plusieurs salles du temple.

Construction d'images en GFA Basic

Avant tout, il faut définir la taille des éléments graphiques et la taille des images construites. Cet article ne traite que de la basse résolution (320x200 pixels en 16 couleurs).
C'est la résolution la plus utilisée, aussi bien pour les jeux que pour le graphisme pur. Les éléments font 16x16 pixels. Ce n'est pas une obligation, mais c'est devenu une norme de *facto*. Il suffit de regarder les images des jeux d'arcades pour en être convaincu. Avec 18x10 éléments, on obtient une image de 288x160 pixels.
C'est un format assez courant. Pour obtenir une image prenant pratiquement tout l'écran, il faut utiliser un format de 20x12 éléments (320x192 pixels), mais il vaut mieux laisser de l'espace pour afficher différents intervenants (décors divers, scores, nom du logiciel, nombre de vies, etc.)
En prenant des éléments de 16x20 pixels, on peut couvrir entièrement l'écran avec un format de 20x12 cases.

Taille d'un élément graphique

Un élément de 16x16 pixels (basse résolution) stocké dans une variable alphanumérique prend 134 octets. La place mémoire nécessaire pour 256 éléments est de 34304 octets.

```
'*********************************
' * CALCUL DE LA TAILLE MEMOIRE  *
' * D'UN ELEMENT DE 16x16 PIXELS *
' *********************************
GET 1,1,16,16,var$
GET 1,1,16,16,var$
PRINT LEN(var$)
```

Stockage des éléments en mémoire

Les instructions graphiques du *GFA Basic* (PUT et GET) stockent les données graphiques dans des variables alphanumériques. Les éléments graphiques sont donc stockés dans un tableau alphanumérique. Une variable nb_elements% permet de connaître le nombre d'éléments en mémoire.

```
DIM element$(256)
nb_elements%=0
```

Pour éviter de perdre de la place en mémoire, les tableaux commencent à l'indice 1 grâce à l'instruction OPTION BASE 1.

Sauvegarde avec l'instruction STORE

A première vue, la technique de sauvegarde la plus simple serait d'utiliser l'instruction STORE qui permet de sauver un tableau alphanumérique avec une seule instruction. Mais cette méthode ne fonctionne pas toujours correctement. L'instruction STORE utilise les codes de contrôles 10, 11 et 27 pour repérer les fins de lignes et la fin du fichier. Lorsque ces codes sont présents dans une chaîne alphanumérique (ce qui arrive souvent avec des données graphiques), l'instruction RECALL qui doit recharger le tableau se trompe de

repère et charge les lignes n'importe comment. En résumé, STORE n'est efficace que pour sauver des textes ASCII normaux (codes commençant à 32).

Sauvegarde avec l'instruction BPUT

L'instruction BPUT permet de sauver une zone de mémoire dans un fichier. Grâce à cette instruction, on peut sauver les éléments graphiques les uns derrière les autres dans un fichier. La procédure Save_element(nom$) du programme GEN_IMGS utilise ce système. Elle sauve au début du fichier la palette de couleurs des éléments. Les données graphiques sont relues avec l'instruction BGET qui permet de lire des données à partir d'un fichier. La procédure Load_element(nom$) du programme AFF_IMGS fonctionne de cette manière.

Stockage des images en mémoire

Pour mémoriser une image, il faut un tableau de 18x10 éléments. S'il y a moins de 256 éléments, ils peuvent être codés sur des octets (symbole |). Une image prend alors 18x10 octets =180 octets. On peut avoir 720x1024/180 images sur une disquette, soit 4096 images.

```
DIM images|(10,18,10)
```

Le tableau images|() peut contenir 10 images. C'est un chiffre aussi valable qu'un autre pour un programme de démonstration, mais cela peut tout aussi bien être 50.

```
DIM images|(50, 18,10)
```

Le *GFA Basic* ne peut pas manipuler des tableaux de taille supérieure à 32 Ko. Le tableau image|() ne peut donc contenir plus de 177 images (32000/180). Il est possible de les répartir sur plusieurs tableaux.

Si il y a plus de 256 éléments possibles, on peut les coder avec des entiers courts (16 bits-symbole &). Il peut alors y avoir 65536 éléments différents. Le codage d'une image prend 360 octets (18x10x2). On peut avoir 720x1024/360 images sur une disquette, soit 2048 images.

```
DIM images&(10, 18,10)
```

Il ne peut y avoir plus de 90 images dans le tableau images&() à cause de la limitation de 32 Ko pour un tableau. Il faut donc stocker les images sur plusieurs tableaux.

Affichage d'une image

Pour afficher une image, il faut afficher chaque élément graphique à sa place. La procédure Aff_image() affiche l'image num_image% à partir de la position graphique (pos_x0%,pos_y0%). On peut afficher une image à n'importe quel endroit de l'écran. Les variables px% et py% contiennent la position d'affichage de l'élément courant. Après l'affichage de chaque élément, on incrémente px% de 16 pixels. De même après l'affichage d'une ligne, on incrémente py% de 16 pixels.

```
' ***************************************************
' *  AFFICHAGE IMAGE num_image% A LA POSITION       *
' *  (pos_x0%,pos_y0%)                              *
' ***************************************************
PROCEDURE Aff_image (num_image%)
  LOCAL ix%,iy%
  LOCAL px%,py
  '
  px%=pos_x0%          ! INIT POS X IMAGE
  py%=pos_y0%          ! INIT PAS Y IMAGE
  FOR iy%=1 TO 10      ! BOUCLE DES LIGNES
    FOR ix%= 1 TO 18   ! BOUCLE DES CASES
      PUT px%, py%, images|(num_image%, ix%, iy%)
      INC px%,16       ! PASSAGE POS X SUIVANTE
    NEXT ix%           ! FIN BOUCLE CASE
    INC py%,16         ! PASSAGE LIGNE SUIVANTE
    px%=pos_x0%        ! RETOUR DEBUT LIGNE
  NEXT iy%             ! FIN BOUCLE LIGNES
 RETURN                ! FIN ROUTINE
```

Sauvegarde des images sur disque

Le tableau contenant les images est sauvé grâce à la fonction Save_tableau(). Cette fonction calcule l'adresse et la taille d'un tableau d'octets avant de sauver le tout dans un fichier binaire. Les informations permettant de comprendre le codage des tableaux sont présentes dans le manuel du *GFA Basic 3.0* au chapitre «Variables et gestion de la mémoire».

La fonction Save_tableau n'est valable que pour des tableaux d'octets. Pour sauver un autre type de tableau, il faut augmenter la taille mémoire des cases (1 octet pour un tableau d'octet, 2 octets pour un tableau d'entier court (&), 4 octets pour un tableau d'entier long (%).

Lecture des images à partir d'un disque

Le tableau des images est chargé en mémoire avec la routine Load_tableau qui relit un tableau sauvé avec la procédure Save_tableau. Cette routine ne peut relire que des tableaux d'octets.

Système de construction d'images complet

Pour avoir un système de construction complet, il faut un utilitaire de création des éléments graphiques, un utilitaire de «fabrication» des images, et un système d'affichage.

Les programmes GEN_ELEM, EDIT_IMG et AFF_IMGS sont des utilitaires écrits en *GFA Basic 3.0*. Ils remplissent chacun une fonction bien précise dans la «chaîne de fabrication» des images.

Programme GEN_ELEM

Ce programme est un convertisseur d'images. Il lit un ensemble d'éléments graphiques stockés sur des images au format graphique *Degas* et génère un fichier binaire réutilisable par d'autres programmes. Les éléments graphiques de base sont dessinés avec le logiciel de dessin *Degas Elite*. Une fois les images terminées, le program-

meur stocke la position de tous les éléments graphiques dans le programme GEN_ELEM grâce à l'instruction DATA. Une fois lancé, le programme lit toutes les images les unes après les autres, et génère le fichier d'éléments graphiques.

Programme EDIT_IMG

Cet utilitaire sert à «fabriquer» les images. L'utilisateur peut l'employer pour les modifier à volonté. Il se présente sous la forme d'une carte avec un menu (sauvegarde des images sur disque, lecture d'un jeu d'images, sortie du programme) en bas à gauche, le dessin d'un élément à droite, et un numéro de carte compris entre 1 et 10 affiché aussi à droite.

Si l'utilisateur clique sur une case de l'image, l'élément de la case est remplacé par l'élément courant. C'est ainsi qu'il est possible de placer n'importe quel élément à n'importe quelle position de l'image.

L'éditeur permet de travailler sur 10 images à la fois. Le numéro de l'image courante est affiché en bas à droite de l'écran. Le programme peut être modifié pour travailler avec n'importe quel nombre d'images.

Programme AFF_IMGS

Ce petit programme est une démonstration du système de construction des images. Il lit un fichier d'images et les affiche les unes après les autres. Pour obtenir un affichage rapide, ce programme utilise une astuce courante dans la programmation des jeux: l'image est construite dans une image de travail situé en mémoire. Lorsque le dessin est terminé, on recopie l'image de travail sur l'écran. L'utilisateur a l'impression d'un affichage instantané, mais ce n'est qu'une illusion.

```
' ***************************************************
' *                                                 *
' *  GENERATION DU FICHIER D'ELEMENTS GRAPHIQUES    *
' *                                                 *
' ***************************************************
' ************************
' *  Programme GEN_ELEM  *
' ************************
' ***************************************************
' *  (C) 1990 Patrick Leclercq & ATARI MAGAZINE     *
' ***************************************************
' ************************************************************
' * Cet utilitaire génère un fichier binaire contenant les   *
' * éléments graphiques a partir du nom d'une image DEGAS,   *
' * et de la liste des positions des éléments (stockées      *
' * dans des lignes de DATA).                                *
' ************************************************************
' ____________________________________________________________
' VARIABLES GLOBALES :
' element$()    : Tableau contenant les éléments graphiques
' nb_elements%  : Nombre éléments dans le tableau element$()
' nom_image$    : Nom image DEGAS contenant les éléments à lire
' nom_fichier$  : Nom fichier contenant les éléments graphiques
' pal_elements$: Palette de couleurs des éléments
' ____________________________________________________________
'
RESERVE 50000          ! RESERVATION DE MEMOIRE POUR BASIC
OPTION BASE 1          ! LES TABLEAUX COMMENCENT A 1
'
DIM element$(256)      ! TABLEAU LES ELEMENTS GRAPHIQUES
nb_elements%=0         ! NOMBRE D'ELEMENTS DANS LE TABLEAU
'
RESTORE
READ nom_image$          !LECTURE DU NOM DE IMAGE A LIRE
READ nom_fichier$        !LECTURE NOM FICHIER BINAIRE A GENERER
@load_degas(nom_image$) !LECTURE DE L'IMAGE DEGAS ELITE
@lecture_elements        !LECTURE ELEMENTS GRAPHIQUES IMAGE
@save_elements(nom_fichier$)!SAUVER DISQUE FICHIER ELEMENTS
END                      !FIN DU PROGRAMME
'
' ____________________________________________
' ******************************
' *  LECTURE IMAGE DEGAS ELITE  *
' ******************************
PROCEDURE load_degas(nom$)
 LOCAL pal$
 '
 pal$=SPACE$(32)          ! RESERVATION MEMOIRE POUR PALETTE
 OPEN "i",#1,nom$         ! OUVERTURE DU FICHIER
 SEEK #1,2                ! ON EVITE L'INDICATEUR RESOLUTION
 BGET #1,VARPTR(pal$),32  ! LECTURE DES 32 OCTETS DE PALETTE
 pal_element$=pal$        ! MEMORISATION PALETTE DES ELEMENTS
 VOID  XBIOS(6,L:VARPTR(pal$))! MODIFICATION PALETTE SYSTEME
 BGET #1,XBIOS(2),32000   ! LECTURE IMAGE
 CLOSE #1                 ! FERMETURE DU FICHIER
RETURN
'
' ____________________________________________
' ****************************************
' *  SAUVEGARDE DES ELEMENTS GRAPHIQUES  *
' ****************************************
' ****************************************************
' *  FORMAT DU FICHIER BINAIRE :                     *
' *  - Palette couleur des éléments (32 octets);     *
' *  - Nombre d'éléments du fichier (1 octet);       *
' *  - Elément graphique 1          (134 octets);    *
' *  - Elément graphique 2          (134 octets);    *
' *  - Elément graphique n          (134 octets);    *
' *  - Dernier élément graphique    (134 octets);    *
' ****************************************************
PROCEDURE save_elements(nom$)
 LOCAL i%
 '
 OPEN "o",#1,nom$                  ! OUVERTURE DU FICHIER
 BPUT #1,VARPTR(pal_element$),32   ! ECRITURE PALETTE ELEMENTS
 OUT #1,nb_elements%               ! ECRITURE NOMBRE ELEMENTS
 FOR i%=1 TO nb_elements%          ! DEBUT BOUCLE LECTURE
   BPUT #1,VARPTR(element$(i%)),134! ECRITURE DE L'ELEMENT i%
 NEXT i%                           ! FIN BOUCLE LECTURE
 CLOSE #1                          ! FERMETURE DU FICHIER
RETURN
'
' ____________________________________________
' ************************************
' *  LECTURE D'UN ELEMENT GRAPHIQUE  *
' ************************************
' ****************************************
' *  num%  : Numéro de l'élément a lire  *
```

```
' *  posx% : Position X de l'élément      *
' *  posy% : Position Y de l'élément      *
' ****************************************
' ************************************************************
' *  L'instruction GET fonctionne avec 4 paramètres :        *
' * x1, y1, x2, y2. La fonction lec_element doit fabriquer
*
' * les coordonnées x2 et y2 à partir des position posx% et   *
' * posy%. Etant donné que les éléments graphiques ont une    *
' * taille de 16 pixels, il faut rajouter (16-1) à la         *
' * coordonnés de début de l'élément pour obtenir la          *
' * coordonnée de fin. Pourquoi (16-1) et non 16 ?            *
' * Le premier pixel compte aussi dans la longueur de         *
' * l'élément. Avec un début de 10, un élément d'une longueur *
' * de 16 pixels se termine au pixel 25. Les pixels 10,11,    *
' * 12,13,..,24,25 constituent cet élément. C'est pourquoi on
*
' * utilise la formule (16-1).                                *
' ************************************************************
PROCEDURE lec_element(num%,posx%,posy%)
  LOCAL posx2%,posy2%
  '
  posx2%=posx%+(16-1)
  posy2%=posy%+(16-1)
  ' LECTURE ELEMENT num%
  GET posx%,posy%,posx2%,posy2%,element$(num%)
RETURN
'
' ____________________________________________
' *********************************************
' *  LECTURE DE TOUS LES ELEMENTS GRAPHIQUES  *
' *********************************************
PROCEDURE lecture_elements
 LOCAL px%,py%
 '
 DO
  READ px%                      ! LECTURE PREMIERE DONNEE
  EXIT IF px%=-1                ! TEST SI DONNEE DE FIN (-1)
  INC nb_elements%              ! UN ELEMENT GRAPHIQUE DE PLUS
  READ py%                      ! LECTURE POSITION Y ELEMENT
  @lec_element(nb_elements%,px%,py%)! LECTURE D'UN ELEMENT
 LOOP
RETURN
'
' ____________________________
' ****************************
' *  DEFINITION DES ELEMENTS  *
' ****************************
' DATA Nom image DEGAS ELITE
' DATA Nom fichier éléments à générer
' DATA x,y
' DATA x,y
' DATA -1   (-1 = INDICATEUR DE FIN DE DATA)
'
' ____________________________________________
DATA DESSIN.PI1
DATA ELEMENTS.ELE
' ____________________________________________
DATA 1,   1
DATA 18,  1
DATA 35,  1
DATA 86,  1
DATA 103, 1
DATA 120, 1
DATA 137, 1
DATA 154, 1
DATA 171, 1
DATA 188, 1
DATA 205, 1
DATA 222, 1
DATA 239, 1
DATA 256, 1
DATA 273, 1
DATA 290, 1
DATA -1


' ***********************************
' *                                 *
' *  EDITEUR D'IMAGES CONSTRUITES   *
' *                                 *
' ***********************************
' ************************
' *  Programme EDIT_IMG  *
' ************************
' ************************************************
' *  (C) 1990 Patrick Leclercq & ATARI MAGAZINE  *
' *  Programmation Interpréteur GFA BASIC 3.03   *
' ************************************************
' ____________________________________________
' ************************************
' *  VARIABLES GLOBALES DU PROGRAMME  *
' ************************************
' element$()        :Tableau contenant les éléments graphiques
' nb_elements%      :Nb éléments graphiques dans element$()
' images|()         :Tableau contenant le codage des images
' zones%()          :Tableau contenant les zones de clics
' nb_zones%         :Nombre de zones dans tableau zones%()
' sortie%           :Condition de sortie du prg
' element_courant%  :Num élément selectionné par utilisateur
' num_image%        :Numéro de l'image en cours d'édition
' posx0%           :Position x d'affichage de l'image courante
' posy0%           :Position y d'affichage de l'image courante
' clic_x%          :Pos x dernier clic dans zone écran
' clic_y%          :Pos y dernier clic dans zone écran
'
' ____________________________________________
RESERVE 50000          !RESERVATION DE MEMOIRE POUR LE BASIC
OPTION BASE 1          !LES INDICES DE TABLEAU COMMENCENT A 1
'
DIM images|(10,18,10) !TABLEAU POUR LES IMAGES
'
DIM element$(256)      !TABLEAU POUR LES ELEMENTS
nb_elements%=0
'
DIM zone%(10,4)        !TABLEAU POUR LES ZONES DE CLICS
nb_zones%=0
'
@main                  !APPEL PROCEDURE PRINCIPALE
END                    !FIN DU PROGRAMME
'
' ____________________________
' ****************************
' *  ATTENTE D'UN CLIC SOURIS  *
' ****************************
PROCEDURE att_clic(VAR xm%,ym%)
```

```
  LOCAL km%
  '
  REPEAT
    MOUSE xm%,ym%,km%
  UNTIL km%=1
RETURN
'
' ____________________
' ****************************
' *  ATTENTE ARRET CLIQUAGE  *
' ****************************
PROCEDURE attOclic
  REPEAT
  UNTIL MOUSEK=0
RETURN
'
' ____________________________________
' ************************************
' *  TEST SI UNE POSITION (px/py)    *
' *  EST DANS UNE ZONE (x1,y1,x2,y2) *
' ************************************
FUNCTION tst_zone(px%,py%,x1%,y1%,x2%,y2%)
 LOCAL r%
 '
 r%=FALSE
 IF ((px%>=x1%) AND (px%<=x2%) AND (py%>=y1%) AND (py%<=y2%))
   r%=TRUE
 ENDIF
 RETURN r%
ENDFUNC
'
' ____________________________________________
' ***********************************************
' *  SAUVEGARDE D'UN TABLEAU D'OCTETS SUR DISQUE  *
' ***********************************************
PROCEDURE save_tableau(nom$,VAR tab|())
 LOCAL nb_dim%
 LOCAL nb_elem%
 LOCAL adr_data%
 LOCAL taille_data%
 '
 nb_dim%=INT{ARRPTR(tab|())+4}       ! NOMBRE DE DIMENSIONS
 adr_data%=LONG{ARRPTR(tab|())}      ! ADRESSE TABLEAU
 nb_elem%=DIM?(tab|())               ! NOMBRE ELEMENTS TABLEAU
 taille_data%=nb_elem%+nb_dim%*4     ! TAILLE DU TABLEAU
 '
 BSAVE nom$,adr_data%,taille_data% ! SAUVEGARDE ZONE TABLEAU
RETURN
'
' ____________________________________
' ***********************************
' *  LECTURE D'UN TABLEAU D'OCTETS  *
' ***********************************
PROCEDURE load_tableau(nom$,VAR tab|())
  LOCAL adr_data%
  '
  adr_data%=LONG{ARRPTR(tab|())} !ADR ZONE MEMOIRE TABLEAU
  BLOAD nom$,adr_data%           !CHARGEMENT TABLEAU
RETURN
'
' ____________________________________________
' ********************************************
' *  INITIALISATION DU TABLEAU DES IMAGES  *
' ********************************************
PROCEDURE init_images
  LOCAL i%,ix%,iy%
  '
  FOR i%=1 TO 10
    FOR ix%=1 TO 18
      FOR iy%=1 TO 10
        images|(i%,ix%,iy%)=1
      NEXT iy%
    NEXT ix%
  NEXT i%
RETURN
'
' ____________________________________
' ************************************
' *  INITIALISATION DES ZONES DE CLIC  *
' ************************************
PROCEDURE init_clic
  LOCAL x1%,y1%,x2%,y2%
  '
  nb_zones%=0
  DO
    READ x1%
    EXIT IF x1%=-1
    INC nb_zones%
    READ y1%
    READ x2%
    READ y2%
    zone%(nb_zones%,1)=x1%
    zone%(nb_zones%,2)=y1%
    zone%(nb_zones%,3)=x2%
    zone%(nb_zones%,4)=y2%
  LOOP
RETURN
'
' ____________________________
' ****************************
' *  DEFINITION DES ZONES DE CLIC  *
' ****************************
' DATA x1, y1, x2, y2
' DATA x1, y1, x2, y2
' DATA -1        (INDICATEUR DE FIN DE DONNEES)
'
DATA 35 , 187, 165, 197
DATA 253, 190, 285, 197
DATA 253, 167, 285, 174
DATA 35,  167, 165, 177
DATA 35,  177, 165, 187
DATA 200, 190, 218, 197
DATA 200, 167, 218, 174
DATA 16,  3,   303, 162
DATA -1
'
' ____________________
' ********************
' *  LECTURE DES IMAGES  *
' ********************
PROCEDURE lecture_images
 LOCAL nom$
 LOCAL path$
 '
 path$=DIR$(0)+"\*.IMG"          !DETERMINATION CHEMIN COURANT
 FILESELECT path$," ",nom$       !APPEL SELECTEUR DE FICHIERS
```

```
 IF nom$<>""                    !TEST SI PRESENCE D'UN NOM
   HIDEM                        !EFFACER SOURIS
   @load_tableau(nom$,images|()!CHARGEMENT TABLEAU IMAGES
   num_image%=1                 !INIT IMAGE COURANTE
   @aff_image                   !AFFICHAGE IMAGE COURANTE
   @aff_num_image               !AFF NUMERO IMAGE COURANTE
   SHOWM                        !AFFICHAGE SOURIS
 ENDIF
RETURN
'
' ___________________________
' ***************************
' *  SAUVEGARDE DES IMAGES  *
' ***************************
PROCEDURE sauver_images
 LOCAL nom$
 LOCAL path$
 '
 path$=DIR$(0)+"\*.IMG"         !DETERMINATION CHEMIN COURANT
 FILESELECT path$," ",nom$      !APPEL SELECTEUR DE FICHIERS
 IF nom$<>""                    !TEST SI NOM DE FICHIER
   HIDEM                        !EFFACER LA SOURIS
   @save_tableau(nom$,images|()!SAUVER LE TABLEAU IMAGES
   SHOWM                        !AFFICHAGE SOURIS
 ENDIF
RETURN
'
' ____________________________________________
' ********************************************
' *  ROUTINE DE LECTURE DES ELEMENTS GRAPHIQUES  *
' ********************************************
PROCEDURE load_elements(nom$)
 LOCAL i%       ! VARIABLE LOCALE POUR BOUCLE LECTURE
 LOCAL pal$
 '
 OPEN "i",#1,nom$                !OUVERTURE DU FICHIER
 pal$=SPACE$(32)
 BGET #1,VARPTR(pal$),32         !LECTURE DE LA PALETTE
 VOID XBIOS(6,L:VARPTR(pal$))    !DECLARATION PALETTE
 nb_elements%=INP(#1)            !LECTURE NOMBRE ELEMENTS
 FOR i%=1 TO nb_elements%        !DEBUT BOUCLE LECTURE
   element$(i%)=SPACE$(134)    ! RESERVE MEMOIRE POUR ELEMENT
   BGET #1,VARPTR(element$(i%)),134!LECTURE D'UN ELEMENT
 NEXT i%                         !FIN BOUCLE LECTURE
 CLOSE #1                        !FERMETURE DU FICHIER
RETURN
'
' ____________________________________
' ************************************
' *  AFFICHAGE D'UN ELEMENT GRAPHIQUE  *
' ************************************
' ********************************************************
' *  La position d'un élément sur l'écran se calcule de  *
' * la manière suivante :                                *
' *  px% = (colonne%-1)*16+posx0%                        *
' *  Le numéro x de l'élément varie de 1 a 18            *
' *  Affichage élément 1 a posx%                         *
' *  Affichage élément 2 a posx%+16                      *
' ********************************************************
PROCEDURE aff_element(colonne%,ligne%)
 LOCAL px%,py%
 LOCAL num_element%
 '
 num_element%=images|(num_image%,colonne%,ligne%)
 px%=(colonne%-1)*16+posx0%
 py%=(ligne%-1)*16+posy0%
 PUT px%,py%,element$(num_element%)
RETURN
'
' ____________________________________
' ************************************
' *  AFFICHAGE DE L'IMAGE COURANTE  *
' ************************************
PROCEDURE aff_image
  LOCAL ix%,iy%
  '
  FOR iy%=1 TO 10
    FOR ix%=1 TO 18
      @aff_element(ix%,iy%)
    NEXT ix%
  NEXT iy%
RETURN
'
' ____________________________________________________
' ****************************************************
' *  AFFICHAGE DE L'ELEMENT COURANT SUR L'ECRAN  *
' ****************************************************
PROCEDURE aff_el_courant
  PUT 201,175,element$(element_courant%)
RETURN
'
' ____________________________________________
' ********************************************
' *  AFFICHAGE DU NUMERO D'IMAGE SUR L'ECRAN  *
' ********************************************
PROCEDURE aff_num_image
 PBOX 254,175,284,189      !EFFACEMENT ZONE D'AFFICHAGE
 TEXT 265,185,num_image%  !AFFICHAGE NUMERO IMAGE COURANTE
RETURN
'
' ____________________________
' ****************************
' *  AFFICHAGE ECRAN EDITION  *
' ****************************
PROCEDURE aff_edit
  CLS                        ! EFFACEMENT DE L'ECRAN
  '
  PBOX 15,2,304,163          ! ZONE DE L'IMAGE
  '
  BOX 35,167,165,197         ! ZONE DES COMMANDES TEXTES
  LINE 35,177,165,177
  LINE 35,187,165,187
  TEXT 50,175,"SAUVER IMAGES"
  TEXT 46,185,"LECTURE IMAGES"
  TEXT 46,195,"SORTIE EDITEUR"
  '
  BOX 200,167,217,197   ! ZONE DE L'ELEMENT COURANT
  LINE 200,174,217,174
  LINE 200,191,217,191
  '                     ! POSITION ELEMENT COURANT : 201,175
  '
  BOX 253,167,285,197   ! ZONE DE L'IMAGE COURANTE
  LINE 253,174,285,174
  LINE 253,190,285,190
  '
  @aff_image            ! AFFICHAGE IMAGE COURANTE
```

```
  @aff_el_courant        ! AFFICHAGE ELEMENT COURANT
  @aff_num_image         ! AFFICHAGE NUMERO IMAGE COURANTE
RETURN
'
' ____________________________________________________________
' ************************************************************
' *  MODIFICATION DE LA CASE SELECTIONNE PAR L'UTILISATEUR  *
' ************************************************************
' ************************************************************
' *  CALCUL POSITION CASE :                                  *
' *  Le calcul de la position de la case cliqué par          *
' * l'utilisateur se fait en deux temps :                    *
' *   - 1) On calcule la position relative de clic par       *
' *        rapport au bord de l'image (on soustrait la       *
' *        position de l'image à la position de clic).       *
' *        px%=clic_x%-posx0%  et  py%=clic_y%-posy0%        *
' *   - 2) On divise le résultat par 16 (taille d'une case)  *
' *        et on ajoute 1 puisque le résultat de la division*
' *        vas de 0 à 17 pour les colonnes à la place de     *
' *        1 à 18, et de 0 à 9 pour les lignes à la place de*
' *        1 à 10.                                           *
' * ________________________________________________________*
' *  TEST D'ECRITURE :                                       *
' *  Avant de modifier le contenu d'une case de l'image, la *
' * routine regarde si cette case ne contient pas déja la    *
' * bonne valeur. Cela évite un accès vidéo inutile.         *
' * L'affichage est plus agréable comme ça.                  *
' ************************************************************
PROCEDURE modif_case
 LOCAL colonne%
 LOCAL ligne%
 LOCAL px%,py%
 LOCAL element%
 '
 px%=clic_x%-posx0%          ! POS X RELATIVE
 py%=clic_y%-posy0%          ! POS Y RELATIVE
 colonne%=INT(px%/16)+1      ! NUM COLONNE TABLEAU
 ligne%=INT(py%/16)+1        ! NUM LIGNE TABLEAU
 '
 element%=images|(num_image%,colonne%,ligne%)
 IF element_courant%<>element%
  images|(num_image%,colonne%,ligne%)=element_courant%
  HIDEM
  @aff_element(colonne%,ligne%)
  SHOWM
 ENDIF
RETURN
'
' ____________________________________________________
' ****************************************************
' *  DECREMENTATION DE L'IMAGE EN COURS D'EDITION  *
' ****************************************************
' ************************************************************
' * Les images vont de 1 à 10. La routine vérifie si         *
' * l'utilisateur tente d'accéder à une image inférieure à 1 *
' ************************************************************
PROCEDURE dec_image
  IF num_image%<>1      ! TEST SI DECREMENTATION POSSIBLE
    DEC num_image%      ! PASSAGE IMAGE INFERIEURE
    HIDEM               ! EFFACER LA SOURIS
    @aff_image          ! AFFICHAGE NOUVELLE IMAGE
    @aff_num_image      ! AFFICHAGE NUMERO NOUVELLE IMAGE
    SHOWM               ! AFFICHAGE SOURIS APRES MODIF ECRAN
  ENDIF
RETURN
'
' ____________________________________________________
' ****************************************************
' *  INCREMENTATION DE L'IMAGE EN COURS D'EDITION  *
' ****************************************************
' ************************************************************
' *  Les images vont de 1 à 10. La routine vérifie si        *
' * l'utilisateur tente d'accéder à une image >10.           *
' ************************************************************
PROCEDURE inc_image
 IF num_image%<10  !TEST SI INCREMENTATION POSSIBLE (IMAGE<10)
   INC num_image%  !INCREMENTATION IMAGE
    HIDEM          !EFFACEMENT SOURIS
    @aff_image     !AFFICHAGE NOUVELLE IMAGE
    @aff_num_image !AFFICHAGE NUMERO NOUVELLE IMAGE
    SHOWM          !AFFICHAGE SOURIS
  ENDIF
RETURN
'
' ____________________________________________
' ********************************************
' *  DECREMENTATION DE L'ELEMENT COURANT  *
' ********************************************
PROCEDURE dec_element
  IF element_courant%<>1
    DEC element_courant%
    HIDEM
    @aff_el_courant
    SHOWM
    @attOclic
  ENDIF
RETURN
'
' ____________________________________________
' ********************************************
' *  INCREMENTATION DE L'ELEMENT COURANT  *
' ********************************************
PROCEDURE inc_element
  IF element_courant%<>nb_elements%
    INC element_courant%
    HIDEM
    @aff_el_courant
    SHOWM
    @attOclic
  ENDIF
RETURN
'
' ____________________________________
' ************************************
' *  TEST DE SORTIE DU PROGRAMME  *
' ************************************
' ************************************************************
' * Cette routine met à jour la condition de sortie (sortie%)*
' * en fonction de la variable de réponse de la boite        *
' * d'alerte.                                                *
' *  sortie% =0 => l'utilisateur ne veut pas quitter le prg; *
' *  sortie% =1 => l'utilisateur veut quitter le prg;        *
' ************************************************************
PROCEDURE tst_sortie
  LOCAL tst%
  LOCAL m$
```

```
  '
  m$="Voulez-vous vraiment|quitter l'éditeur ?"
  ALERT 2,m$,1,"Non|Oui",tst%
  IF tst%=2
    sortie%=1
  ENDIF
RETURN
'
' ____________________________________________
' ******************************************
' *  EXECUTION DES FONCTIONS DE L'EDITEUR  *
' ******************************************
PROCEDURE exec_fonction(ev%)
  SELECT ev%
  CASE 1
    @attOclic
    @tst_sortie
  CASE 2
    @dec_image
  CASE 3
    @inc_image
  CASE 4
    @attOclic
    @sauver_images
  CASE 5
    @attOclic
    @lecture_images
  CASE 6
    @dec_element
  CASE 7
    @inc_element
  CASE 8
    @modif_case
  ENDSELECT
RETURN
'
' ____________________________________
' ************************************
' *  RECHERCHE DE LA ZONE DE CLIC  *
' ************************************
' ************************************************************
' *  Cette routine renvoit le numéro de la zone            *
' * correspondant aux coordonnées px% et py%. Si aucune zone *
' * ne correspond aux coordonnées, elle renvoit 0.          *
' ************************************************************
FUNCTION cherche_zone(px%,py%)
  LOCAL i%
  LOCAL num_zone%
  LOCAL x1%,y1%,x2%,y2%
  '
  num_zone%=0
  '
  FOR i%=1 TO nb_zones%
    x1%=zone%(i%,1)                          ! LECTURE x1 ZONE i%
    y1%=zone%(i%,2)                          ! LECTURE y1 ZONE i%
    x2%=zone%(i%,3)                          ! LECTURE x2 ZONE i%
    y2%=zone%(i%,4)                          ! LECTURE y2 ZONE i%
    IF @tst_zone(px%,py%,x1%,y1%,x2%,y2%) ! TEST ZONE i%
      num_zone%=i%
    ENDIF
  NEXT i%
  '
  RETURN num_zone%
ENDFUNC
'
' ____________________________________________
' ******************************************
' *  ATTENTE SELECTION FONCTION EDITEUR  *
' ******************************************
' ************************************************************
' *  Cette routine attend que l'utilisateur presse sur une   *
' *  zone de clic correspondant a une fonction de l'éditeur. *
' ************************************************************
' *  1 : Sortie du programme                                 *
' *  2 : Clic zone passage image inférieure                  *
' *  3 : Clic zone passage image supérieure                  *
' *  4 : Clic zone sauvegarde images                         *
' *  5 : Clic zone lecture images                            *
' *  6 : Clic zone décrémentation élément courant            *
' *  7 : Clic zone incrémentation élément courant            *
' *  8 : Clic dans l'image. clic_x% et clic_y% contiennent   *
' *      position clic                                       *
' ************************************************************
FUNCTION att_zone_clic
 LOCAL zone_clic%
 '
 REPEAT
   @att_clic(clic_x%,clic_y%)          ! ATTENTE CLIC SOURIS
   zone_clic%=@cherche_zone(clic_x%,clic_y%) ! TEST CLIC ZONE
  UNTIL zone_clic%<>0                          ! ATT CLIC ZONE
RETURN zone_clic%          ! RENVOI NUMERO ZONE DE CLIC
ENDFUNC
'
' ____________________________________
' ************************************
' *  FONCTION PRINCIPALE DE L'EDITEUR  *
' ************************************
PROCEDURE edit
 LOCAL zone_clic%
 '
 sortie%=0                  ! CONDITION DE SORTIE = 0
 DO
   zone_clic%=@att_zone_clic ! ATTENTE D'UN ORDRE
   @exec_fonction(zone_clic%)! EXECUTION DE LA FONCTION
   EXIT IF sortie%=1         ! TEST SI CONDITION DE SORTIE =1
LOOP
RETURN
'
' ____________________________
' ****************************
' *  PROGRAMME PRINCIPAL  *
' ****************************
PROCEDURE main
 posx0%=16                  ! POSITION X DE L'IMAGE
 posy0%=3                   ! POSITION Y DE L'IMAGE
 num_image%=1               ! INIT NUMERO IMAGE
 element_courant%=1         ! INIT NUMERO ELEMENT COURANT
 @init_clic                 ! INITIALISATION DES ZONES DE CLIC
 @init_images               ! INITIALISATION IMAGES DE DEBUT
 @load_elements("ELEMENTS.ELE") ! CHARGEMENT ELEMENTS
 @aff_edit                  ! AFFICHAGE DE L'ECRAN D'EDITION
 SHOWM                      ! AFFICHAGE SOURIS
 @edit                      ! PROCEDURE D'EDITION
RETURN

' ****************************************
```

```
' *                                    *
' *  AFFICHAGE DES IMAGES CONSTRUITES  *
' *                                    *
' ****************************************
' **************************
' *   programme AFF_IMGS   *
' **************************
' ******************************************
' *  (C) Patrick Leclercq & ATARI MAGAZINE  *
' ******************************************
' ____________________________________________
RESERVE 50000          ! RESERVATION MEMOIRE POUR LE BASIC
'
OPTION BASE 1          ! LES INDICES DE TABLEAU COMMENCENT A 1
'
DIM images|(10,18,10) ! TABLEAU POUR LES IMAGES
DIM element$(256)      ! TABLEAU POUR LES ELEMENTS
nb_elements%=0         ! NOMBRE ELEMENTS EN MEMOIRE
'
ecran_travail%=MALLOC(32000) ! 32000 OCTETS POUR ECRAN TRAVAIL
@main                        ! APPEL ROUTINE PRINCIPALE
VOID MFREE(ecran_travail%)   ! LIBERATION RAM ECRAN TRAVAIL
END                          ! FIN DU PROGRAMME
'
' ____________________________________________
' ************************************************
' *  DEFINITION DE L'ECRAN DE TRAVAIL DU SYSTEME  *
' ************************************************
PROCEDURE def_ecr_log(ecran_logique%)
  VOID XBIOS(5,L:ecran_logique%,L:-1,-1)
RETURN
'
' ____________________________
' ***********************************
' *  LECTURE D'UN TABLEAU D'OCTETS  *
' ***********************************
PROCEDURE load_tableau(nom$,VAR tab|())
  LOCAL adr_data%
  '
  adr_data%=LONG{ARRPTR(tab|())}
  BLOAD nom$,adr_data%
RETURN
'
' ____________________________________
' ************************************
' *  LECTURE DES ELEMENTS GRAPHIQUES  *
' ************************************
PROCEDURE load_elements(nom$)
 LOCAL i%
 LOCAL pal$
 '
 OPEN "i",#1,nom$                  ! OUVERTURE DU FICHIER
 pal$=SPACE$(32)
 BGET #1,VARPTR(pal$),32           ! LECTURE DE LA PALETTE
 VOID XBIOS(6,L:VARPTR(pal$))      ! DECLARATION PALETTE
 nb_elements%=INP(#1)              ! LECTURE NOMBRE ELEMENTS
 FOR i%=1 TO nb_elements%          ! DEBUT BOUCLE LECTURE
   element$(i%)=SPACE$(134)        ! RESERVATION MEMOIRE
   BGET #1,VARPTR(element$(i%)),134 ! LECTURE ELEMENT i%
 NEXT i%                           ! FIN BOUCLE LECTURE
 CLOSE #1                          ! FERMETURE DU FICHIER
RETURN
'
' ____________________________________
' ****************************************
' *  AFFICHAGE D'UN ELEMENT DE L'IMAGE  *
' ****************************************
' ************************************************************
' *  image%   : Numéro de l'image à afficher                 *
' *  colonne% : Numéro de la colonne de l'élément (1 à 18)   *
' *  ligne%   : Numéro de la ligne de l'élément   (1 à 10)   *
' ************************************************************
PROCEDURE aff_element(image%,colonne%,ligne%)
 LOCAL px%,py%
 LOCAL num_element%
 '
 ' LECTURE NUM ELEMENT
 num_element%=images|(image%,colonne%,ligne%)
 ' CALCUL POS X ELEMENT
 px%=(colonne%-1)*16+posx0%
 ' CALCUL POS Y ELEMENT
 py%=(ligne%-1)*16+posy0%
 ' AFFICHACHE ELEMENT
 PUT px%,py%,element$(num_element%)
RETURN
'
' ____________________________
' ***************************
' *  AFFICHAGE D'UNE IMAGE  *
' ***************************
' ************************************************************
' * L'affichage ne se fait pas sur l'écran physique (l'écran*
' * l'écran visible), mais sur un autre écran situé en      *
' * mémoire (ecran_travail%). Une fois l'affichage          *
' * complètement terminée, le programme recopie l'écran de  *
' * travail sur l'écran physique grâce à la fonction Bmove. *
' * Cette méthode permet d'obtenir un affichage ultra-rapide*
' * plus agréable à l'oeil.                                 *
' ************************************************************
PROCEDURE aff_image(num_image%)
 LOCAL ix%,iy%
 '
 @def_ecr_log(ecran_travail%)! ECRAN TRAVAIL = ECRAN LOGIQUE
 CLS                         ! EFFACEMENT DE L'ECRAN LOGIQUE
 FOR iy%=1 TO 10
  FOR ix%=1 TO 18
    @aff_element(num_image%,ix%,iy%)! AFFICHAGE D'UN ELEMENT
  NEXT ix%
 NEXT iy%
 @def_ecr_log(XBIOS(2))      ! ECRAN PHYSIQUE = ECRAN LOGIQUE
 BMOVE ecran_travail%,XBIOS(2),32000 ! AFF ECRAN DE TRAVAIL
RETURN
'
' ____________________________________________
' *************************
' *  PROGRAMME PRINCIPAL  *
' *************************
PROCEDURE main
 LOCAL i%
 '
 posx0%=16                          ! INIT POS X IMAGE
 posy0%=3                           ! INIT POSI Y IMAGE
 @load_elements("ELEMENTS.ELE")     ! CHARGEMENT ELEMENTS
 @load_tableau("IMAGES.IMG",images|()) ! CHARGE CODAGE IMAGES
 FOR i%=1 TO 10
   @aff_image(i%)                   ! AFFICHAGE IMAGE i%
   VOID INP(2)                      ! ATTENTE CLAVIER
 NEXT i%
RETURN
```

FABRICATION D'IMAGES PAR ASSEMBLAGE DE SPRITES

Un légo informatique

Voici une technique permettant de dessiner des images à partir de sprites qui est beaucoup utilisée par les développeurs de jeux vidéo pour fabriquer des images avec peu de place mémoire.

Qu'est-ce qu'un sprite ?

Un sprite est une image pouvant s'afficher en superposition sur un fond graphique quelconque. Contrairement à une image rectangulaire affichée avec l'instruction graphique PUT, un sprite peut avoir n'importe quelle forme.
Les sprites peuvent être utilisés pour réaliser des animations divers (monstres de jeux vidéo, vaisseaux spatiaux, lettres se déplaçant sur l'écran, etc.) Ils peuvent aussi être utilisés pour dessiner des images à partir d'éléments graphiques de base.

Affichage d'un sprite

Le *GFA Basic* dispose d'une instruction SPRITE, mais celle-ci est tellement peu performante qu'elle ne présente aucun intérêt pour nous. Elle ne peut afficher que des sprites dessinés en deux couleurs et ne pouvant faire plus de 16x16 pixels. Pour afficher un sprite en 16 couleurs et de dimensions quelconque, il faut utiliser les opérateurs graphiques logiques.
Un sprite est constitué de deux images: la forme et le masque. La forme contient le dessin du sprite réalisé sur un fond de couleur 0. Le masque est un dessin en couleur 0 réalisé sur un fond de couleur 15. Les pixels en couleur 0 du masque correspondent aux pixels affichables de la forme. Les pixels en couleur 15 du masque correspondent aux pixels non affichables de la forme.
Le masque doit être affiché avant la forme. Le masque est affiché avec l'opérateur graphique logique AND et la forme avec l'opérateur graphique logique OR.
L'instruction PUT peut afficher une image avec un opérateur graphique logique. Le type de l'opération logique doit être précisé après la variable alphanumérique (1 pour opérateur AND et 7 pour opérateur OR).

```
PUT px,py,masque$,1  ! Opérateur AND
PUT px,py,forme$,7   ! Opérateur OR
```

Un article présentant les opérateurs graphiques logiques et quelques exemples d'utilisations dans ce numéro.

Génération d'un sprite

Pour avoir un sprite, il faut dessiner le masque et la forme. Le dessin du masque est une opération délicate. C'est souvent la hantise des graphistes. On peut simplifier les opérations en générant le masque et la forme à partir d'une seule image. Cette image doit être dessinée sur un fond de couleur 15. On dit alors que la couleur 15 est une couleur transparente, puisque les pixels en couleur 15 du sprite ne seront pas affichés sur l'écran. Puisque la couleur 15 n'est pas affichée sur l'écran, les sprites ne sont pas dessinés en 16 couleurs, mais en 15 (de la couleur 0 à la couleur 14).

Table des couleurs VDI

Avant d'étudier la transformation d'une image en masque et sprite, il faut étudier la manière dont le *GFA Basic* gère les numéros de couleurs. Les fonctions graphiques du *GFA Basic* n'utilisent pas le codage normal des couleurs, mais le codage couleurs des fonctions graphiques *VDI* (implantés dans les ROMS du ST).
Par exemple, le numéro de couleur VDI correspond à la couleur 15. Lorsqu'une fonction graphique du *GFA Basic* dessine un pixel en couleur 1, elle écrit dans la mémoire vidéo la valeur 15 et non la valeur 1.

Numéro Couleur GEM/VDI	Numéro véritable couleur	Numéro couleur GEM/VDI	Numéro véritable couleur
0	0	8	7
1	15	9	8
2	1	10	9
3	2	11	10
4	4	12	12
5	6	13	14
6	3	14	11
7	5	15	13

Le codage de couleur VDI n'est pas utilisé par tous les programmes ST. Les logiciels de dessin graphiques (*Degas Elite* par exemple) utilisent les vrais numéros de couleurs. Les opérateurs graphiques logiques qui agissent sur les couleurs au niveau codage binaire ont besoin des vrais numéros de couleurs. Pour obtenir la couleur 15 dans la mémoire écran, il faut utiliser la couleur 1 en *GFA Basic*.

Fabrication d'une forme de sprite

L'image de base du sprite est le dessin du sprite réalisé sur un fond de couleur 15. La forme est le dessin du sprite réalisé sur un fond de couleur 0. Pour générer une forme à partir d'une image de base, il faut changer tous les pixels de couleur 15 en couleur 0. Pour le *GFA Basic*, en tenant compte du transcodage de couleur, il faut convertir tous les pixels de couleur 1 en couleur 0.
La procédure @gen_forme fabrique une forme à partir d'une image de base. L'image de base est située à la position (px%,py%). Elle mesure tx% pixels de large et ty% pixels de haut.

```
' ******************************
' *  FABRICATION FORME SPRITE  *
' ******************************
PROCEDURE gen_forme(px%,py%,tx%,ty%)
  LOCAL xpos%,ypos%
  '
  FOR ypos%=py% TO py%+ty%-1
    FOR xpos%=px% TO px%+tx%-1
      IF POINT(xpos%,ypos%)=1
        COLOR 0
        PLOT xpos%,ypos%
      ENDIF
    NEXT xpos%
  NEXT ypos%
RETURN
```

Fabrication d'un masque de sprite

L'image de base du sprite est le dessin du sprite réalisé sur un fond de couleur 15. Le masque est un dessin en couleur 0 réalisé sur un fond de couleur 15. Pour générer un masque à partir d'une image de base, il faut changer tous les pixels de couleur autre que 15 en couleur 0. En tenant compte du codage couleur du *GFA Basic*, il faut convertir tous les pixels de couleur autre que 1 en pixels de couleur 0.
La procédure @gen_masque fabrique un masque de sprite à partir d'une image de base située à la position (px%,py%). Elle mesure px% pixels de large et py% pixels de haut.

```
' ******************************
' *  FABRICATION MASQUE SPRITE  *
' ******************************
PROCEDURE gen_masque(px%,py%,tx%,ty%)
  LOCAL xpos%,ypos%
  '
  FOR ypos%=py% TO py%+ty%-1
    FOR xpos%=px% TO px%+tx%-1
      IF POINT(xpos%,ypos%)<>1
        COLOR 0
        PLOT xpos%,ypos%
      ENDIF
    NEXT xpos%
  NEXT ypos%
RETURN
```

Le programme GENSPRIT

Le programme GENSPRIT fabrique des sprites (masques et formes) à partir d'une image *Degas Elite* et les stocke dans un fichier disque. Le contenu de ce fichier peut être utilisé par divers autres programmes.

```
' ******************************
' *    GENERATEUR DE SPRITES   *
' ******************************
' *   Programme GENSPRIT.GFA   *
' ******************************
' *  (C) 1991 Atari Magazine   *
' *  (C) 1991 Patrick Leclercq *
' ******************************
RESERVE 50000
OPTION BASE 1
'
DIM sprite$(255,2)
' sprite$(n%,1) = Masque sprite n%
' sprite$(n%,2) = Forme  sprite n%
'
@main
END
'
' -------------------------
' *************************
' *  PROCEDURE PRINCIPALE  *
' *************************
PROCEDURE main
  @gen_sprites
  @sauve_sprites("SPRITES.BIN")
RETURN
'
' ---------------------------------
' *********************************
' *  CHARGEMENT IMAGE DEGAS ELITE  *
' *********************************
' ***************************************
' *  Cette procédure stocke la palette  *
' *  de couleur de l'image DEGAS dans   *
' *  la variable globale palette$.      *
' ***************************************
PROCEDURE load_degas(nom$)
  palette$=SPACE$(32)                 ! RESERVE MEM
  OPEN "i",#1,nom$                    ! OUVRIR FICHIER
  SEEK #1,2                           ! SAUTER ENTETE
  BGET #1,VARPTR(palette$),32         ! CHARGE PALETTE
  VOID XBIOS(6,L:VARPTR(palette$))    ! MODIF PALETTE
  BGET #1,XBIOS(2),32000              ! CHARGE IMAGE
  CLOSE #1                            ! FERMER FICHIER
```

```
RETURN
'
' ------------------------------
' ******************************
' *  FABRICATION MASQUE SPRITE  *
' ******************************
PROCEDURE gen_masque(px%,py%,tx%,ty%)
 LOCAL xpos%,ypos%
 '
 FOR ypos%=py% TO py%+ty%-1   ! BOUCLE EN Y
  FOR xpos%=px% TO px%+tx%-1  ! BOUCLE EN X
    IF POINT(xpos%,ypos%)<>1  ! TEST COULEUR <>15
      COLOR 0                 ! FIXE COULEUR PIXEL
      PLOT xpos%,ypos%        ! TRACE PIXEL
    ENDIF
  NEXT xpos%
 NEXT ypos%
RETURN
'
' ------------------------------
' ******************************
' *  FABRICATION FORME SPRITE  *
' ******************************
PROCEDURE gen_forme(px%,py%,tx%,ty%)
 LOCAL xpos%,ypos%
 '
 FOR ypos%=py% TO py%+ty%-1   ! BOUCLE EN Y
  FOR xpos%=px% TO px%+tx%-1  ! BOUCLE EN X
    IF POINT(xpos%,ypos%)=1   ! TEST SI COULEUR 15
      COLOR 0                 ! FIXE COULEUR PIXEL
      PLOT xpos%,ypos%        ! TRACE PIXEL
    ENDIF
  NEXT xpos%
 NEXT ypos%
RETURN
'
' ------------------------------------
' ************************************
' *  SAISIE D'UN SPRITE SUR L'ECRAN  *
' ************************************
PROCEDURE saisie_sprite(n%,px%,py%,tx%,ty%)
 LOCAL image$
 LOCAL px2%,py2%
 '
 px2%=px%+tx%-1
 py2%=py%+ty%-1
 GET px%,py%,px2%,py2%,image$    ! MEMORISE IMAGE
 '
 @gen_masque(px%,py%,tx%,ty%)    ! FABRIC MASQUE
 GET px%,py%,px2%,py2%,sprite$(n%,1) ! SAISIE MASQUE
 '
 PUT px%,py%,image$              ! RESTAURE IMAGE
 @gen_forme(px%,py%,tx%,ty%)     ! FABRIC FORME
 GET px%,py%,px2%,py2%,sprite$(n%,2) ! SAISIE FORME
RETURN
'
' ----------------------------
' ****************************
' *  GENERATION DES SPRITES  *
' ****************************
PROCEDURE gen_sprites
  LOCAL nom_image$
  LOCAL n%
  LOCAL px%,py%
  LOCAL tx%,ty%
  '
  DO
    READ nom_image$             ! LIRE NOM IMAGE
    EXIT IF nom_image$="*FIN"   ! TEST SI FIN
    @load_degas(nom_image$)     ! CHARGE IMAGE
    DO
      READ n%                   ! LIRE NUMERO SPRITE
      EXIT IF n%=-1             ! TEST SI FIN SPRITE
      READ px%                  ! LIRE POS X SPRITE
      READ py%                  ! LIRE POS Y SPRITE
      READ tx%                  ! LIRE TX SPRITE
      READ ty%                  ! LIRE TY SPRITE
      INC nb_sprites%           ! UN SPRITE DE PLUS
      @saisie_sprite(n%,px%,py%,tx%,ty%)
    LOOP
  LOOP
RETURN
'
' ---------------------------------------
' ***************************************
' *  SAUVEGARDE DES SPRITES SUR DISQUE  *
' ***************************************
PROCEDURE sauve_sprites(nom$)
 LOCAL n%
 LOCAL taille%
 LOCAL adr_masque%
 LOCAL adr_forme%
 '
 OPEN "o",#1,nom$
 BPUT #1,VARPTR(palette$),32   !SAUVER PALETTE
 '
 FOR n%=1 TO 255
   taille%=LEN(sprite$(n%,1))   !TAILLE IMAGE
   IF taille%<>0                !PRESENCE SPRITE ?
     BPUT #1,VARPTR(n%),4       !SAUVER NUM SPRITE
     BPUT #1,VARPTR(taille%),4  !SAUVER TAILLE
     adr_masque%=VARPTR(sprite$(n%,1))  !ADR MASQUE
     BPUT #1,adr_masque%,taille%   !SAUVER MASQUE
     adr_forme%=VARPTR(sprite$(n%,2))   !ADR FORME
     BPUT #1,adr_forme%,taille%       !SAUVER FORME
   ENDIF
 NEXT n%
 '
 n%=0                   ! INDICE FIN DE SPRITE
 BPUT #1,VARPTR(n%),4   !SAUVER INDICE FIN DE SPRITE
 '
 CLOSE #1               !FERMETURE FICHIER
RETURN
'
```

```
' ------------------------------------
' ************************************
' *  FORMAT DE DEFINITION DES SPRITES  *
' ************************************
' -------------
DATA IMAGE1.PI1
DATA 1,0,0,32,20
DATA 2,34,0,21,20
DATA 3,61,0,9,44
DATA 4,76,0,72,18
DATA -1
'
' -------------
DATA IMAGE2.PI1
DATA 5,158,0,33,41
DATA 6,201,0,21,27
DATA 7,232,0,40,47
DATA -1
'
' -------------
DATA IMAGE3.PI1
DATA 8,277,0,9,10
DATA 9,292,0,12,17
DATA -1
'
' -------
DATA *FIN
```

Ce programme peut générer des sprites à partir de plusieurs images *Degas Elite*. Le nom des images est stocké dans les lignes de DATA avant la position des sprites. La valeur -1 indique la fin des sprites de l'image courante. La chaîne de caractères *FIN indique qu'il n'y a plus d'images à charger et donc plus de sprites.
Les sprites sont dessinés sur un fond de couleur 15. Ils sont définis dans le programme sous la forme d'un numéro, d'une position PX, d'une position PY, d'une largeur TX et d'une hauteur TY. Le numéro est le numéro de référence du sprite. Les numéros de référence n'ont pas besoin de se suivre. Il est possible de définir le sprite 12 à la suite du sprite 6. Cette technique permet d'insérer un nouveau sprite dans la liste sans être obligé de renuméroter tous les sprites suivants.

Fonctionnement de GENSPRIT

Les sprites sont stockés dans le tableau sprite$(). Le masque du sprite n% est stocké dans la variable sprite$(n%,1) et la forme dans la variable sprite$(n%,2). Le tableau sprite$() est déclaré avec 255 éléments, mais peut être agrandi à volonté.
La routine @load_degas charge une image *Degas Elite* sur l'écran. A la différence des routines de chargement présentés habituellement dans cette rubrique, la palette de couleur de l'image n'est pas stockée dans une variable locale, mais dans la variable globale palette$. Le contenu de cette variable est utilisée par la procédure @sauve_sprites.
La procédure @gen_masque(px%,py%,tx%,ty%) fabrique un masque à partir de l'image située à la position (px%,py%) et de dimensions (tx%,ty%). La procédure @gen_forme(px%,py%,tx%,ty%) fabrique une forme de sprite à partir de l'image située à la position (px%,py%) et de dimension (tx%,ty%).
La procédure @saisie_sprite fabrique le masque et la forme d'un sprite et sauve le tout en mémoire. Le dessin du masque modifie l'image originale du sprite, c'est pourquoi @saisie_sprite sauve dans image$ l'image du sprite et la réaffiche après avoir généré le masque.
La routine @gen_sprite supervise la fabrication des sprites. Elle charge les images *Degas Elite*, lit les données des sprites et appelle les procédures de fabrications du masque et de la forme. C'est la routine la plus importante du programme.

Sauvegarde des sprites dans un fichier disque

La procédure @sauve_sprites(nom$) sauve les sprites dans un fichier disque portant le nom nom$. Le fichier commence par 32 octets contenant la palette de couleurs utilisée par les sprites, suivis du stockage des sprites. Les sprites sont stockés sous la forme du numéro de sprite (4 octets), la taille d'un élément graphique (4 octets - taille du masque et de la forme), du masque et de la forme. La fin du fichier est signalée par la valeur 0 stockée sur 4 octets.

```
BPUT #1,VARPTR(n%),4
BPUT #1,VARPTR(taille%),4
```

Les variables n% et taille% sont sauvées sous la forme d'une zone mémoire de 4 octets car elles occupent 4 octets en mémoire. La fonction VARPTR(var) fournis l'adresse de la variable var (quelque soit le type de variable).

Amélioration de GENSPRIT

Il est possible de stocker les coordonnées des sprites dans un fichier texte et de modifier GENSPRIT pour qu'il extraie les informations dont il a besoin à partir de ce fichier.
Cela évite de modifier le programme pour générer un nouveau fichier de sprites. Cela permet aussi d'avoir une version compilée de GENSPRIT ne prenant que peu de place disque et s'exécutant plus rapidement que la version interprété. Le fichier texte pourrait ressembler à ceci:

```
; ************************
; *  DEFINITION SPRITES  *
; ************************
IMAGE1.PI1
1 0  0 32 20
2 34 0 21 20
3 61 0 9 44
4 76 0 72 18
IMAGE2.PI1
5 158 0 33 41
6 201 0 21 27
7 232 0 40 47
IMAGE3.PI1
8 277 0 9 10
9 292 0 12 17
*FIN
```

Vous pouvez aussi ajoutez à GENSPRIT la possibilité de sauver des sprites dans un fichier qui en contient déja, évitant ainsi de redessiner des sprites déja faits.
Il est possible de stocker le numéro et le nombre d'octets d'un sprite dans une variable de type entier 16 bits (symbole &). Cela représente une économie de 4 octets par sprites. Mais il faut faire attention à ne pas mélanger les symboles % et & dans le corps du programme.

```
BPUT n&,VARPTR(n&),2
BPUT taille&,VARPTR(taille&),2
```

Si le nombre de sprites ne risque pas de dépasser 255, il est même possible de stocker le numéro de sprite dans une variable du type byte (symbole |). Cette variable n'occupe qu'un octet. La sauvegarde de cet octet peut se faire avec l'instruction BPUT, ou avec l'instruction OUT.

```
BPUT n|,VARPTR(n|),1
OUT #1,n|
```

Utiliser des variables de différents types en *GFA Basic* est risqué, car il est facile de changer un symbole & en %, ou | en %, voire même d'oublier complètement le symbole de type. Ce genre d'erreur est toujours très difficile à repérer. Pour éviter cela, il est préférable d'utiliser un type unique de variable et de s'y tenir. Si l'utilisation d'une variable particulière vous permet d'optimiser votre programme (gain de place mémoire ou de vitesse d'exécution), il faut mettre de nombreux commentaires dans votre programme, de manière à ne pas oublier que cette variable est différente des autres.

Fabrication d'une image par assemblage de sprites

Le programme DRAWIMG fabrique des images en affichant les uns par dessus les autres plusieurs sprites. Ces sprites sont chargés à partir d'un fichier disque généré par le programme GENSPRIT.

```
' ******************************
' *    CONSTRUCTION D'IMAGES   *
' *     A PARTIR DE SPRITES    *
' ******************************
' *   Programme DRAWIMG.GFA    *
' ******************************
' *  (C) 1991 Atari Magazine   *
' *  (C) 1991 Patrick Leclercq *
' ******************************
RESERVE 100000
OPTION BASE 1
'
' ------------------------------
' ******************************
' *  MEMORISATION DES SPRITES  *
' ******************************
DIM sprite$(200,2)
' sprite$(n%,1) = Masque sprite n%
' sprite$(n%,2) = Forme  sprite n%
'
' ------------------------------
' ******************************
' *  MEMORISATION DES IMAGES   *
' ******************************
DIM image%(50,40,3)
DIM taille_image%(50)
' image%(img%,n%,1) = Num sprite n%   - image img%
' image%(img%,n%,2) = Pos X sprite n% - image img%
' image%(img%,n%,3) = Pos Y sprite n% - image img%
'
@main
END
'
' --------------------------
' **************************
' *  PROCEDURE PRINCIPALE  *
' **************************
PROCEDURE main
  @charge_sprites("SPRITES.BIN")
  @init_images
  @demo_images
RETURN
'
' ------------------------------------
' ************************************
' *  DEMONSTRATION CONSTRUCTION IMAGES  *
' ************************************
PROCEDURE demo_images
  CLS
  DO
    LOCATE 1,1
    INPUT "Num image (0 pour fin) ",img%
    EXIT IF img%=0
    CLS
    PRINT img%
    @draw_image(img%,100,100)
  LOOP
RETURN
'
' ---------------------------
' ***************************
' *  AFFICHAGE D'UN SPRITE  *
' *  DU TABLEAU sprite$()   *
' ***************************
' *  n%  : Numéro sprite    *
' *  px% : Pos X sprite     *
' *  py% : Pos Y sprite     *
' ***************************
PROCEDURE aff_sprite(n%,px%,py%)
  PUT px%,py%,sprite$(n%,1),1    ! AFFICHAGE MASQUE
  PUT px%,py%,sprite$(n%,2),7    ! AFFICHAGE FORME
RETURN
'
' --------------------------------
' ********************************
' *   AFFICHAGE DE L'IMAGE img%  *
```

```
' ********************************
' * img%  : Numéro image         *
' * posx% : Position X affichage *
' * posy% : Position Y affichage *
' ********************************
PROCEDURE draw_image(img%,posx%,posy%)
  LOCAL n%
  LOCAL sprite%
  LOCAL px%,py%
  '
  FOR n%=1 TO taille_image%(img%)
    sprite%=image%(img%,n%,1)     ! NUMERO SPRITE
    px%=image%(img%,n%,2)+posx%   ! PX SPRITE
    py%=image%(img%,n%,3)+posy%   ! PY SPRITE
    @aff_sprite(sprite%,px%,py%) ! AFF SPRITE
  NEXT n%
RETURN
'
' ---------------------------
' ****************************
' *  CHARGEMENT DES SPRITES  *
' *    A PARTIR DU DISQUE    *
' ****************************
' *  Les sprites sont lus à  *
' *  partir du fichier nom$  *
' ****************************
PROCEDURE charge_sprites(nom$)
 LOCAL pal$
 LOCAL n%
 LOCAL taille%
 LOCAL adr_masque%
 LOCAL adr_forme%
 '
 OPEN "I",#1,nom$              ! OUVERTURE FICHIER
 '
 pal$=SPACE$(32)               ! RESERVE 32 OCTETS
 BGET #1,VARPTR(pal$),32       ! CHARGE PALETTE
 VOID XBIOS(6,L:VARPTR(pal$)) ! CHANGE PALETTE
 '
 DO
  BGET #1,VARPTR(n%),4           ! CHARGE NUM SPRITE
  EXIT IF n%=0                   ! TEST FIN FICHIER
  BGET #1,VARPTR(taille%),4    ! CHARGE TAILLE SPRITE
  sprite$(n%,1)=SPACE$(taille%) ! RESERVE MEM MASQUE
  sprite$(n%,2)=SPACE$(taille%) ! RESERVE MEM FORME
  adr_masque%=VARPTR(sprite$(n%,1)) ! ADR MASQUE
  adr_forme%=VARPTR(sprite$(n%,2))  ! ADR FORME
  BGET #1,adr_masque%,taille%   ! CHARGE MASQUE
  BGET #1,adr_forme%,taille%    ! CHARGE FORME
 LOOP
 '
 CLOSE #1                       ! FERMETURE FICHIER
RETURN
'
' ------------------------------
' ******************************
' *  INITIALISATION DES IMAGES  *
' ******************************
PROCEDURE init_images
 LOCAL num_image%
 LOCAL nb_sprites%
 LOCAL n%,px%,py%
 '
 RESTORE def_images
 DO
  READ num_image%
  EXIT IF num_image%=0
  nb_sprites%=0
  DO
    READ n%
    EXIT IF n%=-1
    READ px%,py%
    INC nb_sprites%
    image%(num_image%,nb_sprites%,1)=n%
    image%(num_image%,nb_sprites%,2)=px%
    image%(num_image%,nb_sprites%,3)=py%
  LOOP
  taille_image%(num_image%)=nb_sprites%
 LOOP
RETURN
'
' ---------------------------
' ***************************
' *  DEFINITION DES IMAGES  *
' ***************************
' DATA num sprite,px,py
' DATA num sprite,px,py
' DATA -1
def_images:
' ********************
' *  PREMIERE IMAGE  *
' ********************
DATA 1
DATA 1,10,10
DATA 2,20,20
DATA 3,40,40
DATA -1
'
' *******************
' *  SECONDE IMAGE  *
' *******************
DATA 2
DATA 4,10,10
DATA 5,15,15
DATA 6,30,30
DATA -1
'
' *********************
' *  TROISIEME IMAGE  *
' *********************
DATA 3
DATA 2,0,0
DATA 7,3,3
DATA 8,28,37
```

```
DATA 9,20,20
DATA -1
'
' *********************
' *  QUATRIEME IMAGE  *
' *********************
DATA 4
DATA 2,0,0
DATA 7,5,5
DATA 1,2,2
DATA -1
'
' *******************
' *  FIN DE DONNES  *
' *******************
DATA 0
```

Chaque image est définie par une série de sprites. Ceux-ci sont eux-même définis par un numéro de référence et une position d'affichage. Ils sont affichés dans l'ordre d'appararition: le premier sprite est affiché en premier, suivi du second, etc. La position d'affichage est relative au bord de l'image. L'image peut être affichée à n'importe quel endroit de l'écran. La valeur -1 signale la fin d'une image. La valeur 0 signale que la dernière image vient d'être définie. Les numéros des sprites sont les numéros de références qui ont été assignés aux sprites dans les instructions DATA du programme GENSPRIT.

Fonctionnement de DRAWIMG

Les sprites sont stockés dans le programme DRAWIMG de la même manière que dans le programme GENSPRIT, c'est-à-dire dans le tableau sprite$(). La procédure @charge_sprites(nom$) charge les sprites du fichier nom$ dans le tableau sprite$(). La valeur 0 est le signe de fin de fichier. La taille et le numéro des sprites qui sont stockés sous la forme de 4 octets sont lus avec l'instruction BGET.

```
PROCEDURE charge_sprites(nom$)
  LOCAL pal$
  LOCAL n%
  LOCAL taille%
  LOCAL adr_masque%
  LOCAL adr_forme%
  '
  OPEN "I",#1,nom$
  '
  pal$=SPACE$(32)
  BGET #1,VARPTR(pal$),32
  VOID XBIOS(6,L:VARPTR(pal$))
  '
  DO
    BGET #1,VARPTR(n%),4
    EXIT IF n%=0
    BGET #1,VARPTR(taille%),4
    sprite$(n%,1)=SPACE$(taille%)
    sprite$(n%,2)=SPACE$(taille%)
    adr_masque%=VARPTR(sprite$(n%,1))
    BGET #1,adr_masque%,taille%
    adr_forme%=VARPTR(sprite$(n%,2))
    BGET #1,adr_forme%,taille%
  LOOP
  !
  CLOSE #1
RETURN
```

La procédure ouvre le fichier en lecture, charge les 32 octets de la palette de couleur des sprites, puis utilise la fonction système XBIOS(6) pour remplacer la palette l'écran par la palette des sprites.
La procédure @init_images charge la définition des images à partir des DATA. Les images sont stockées dans le tableau image%(). Le tableau taille_image%(img%) contient le nombre de sprites de l'image img%. La routine @init_images utilise une double boucle DO-LOOP. La boucle interne s'occupe de lire la position des sprites définissant une image. La sortie de la boucle est assurée par une instruction EXIT IF lorsque le programme rencontre la valeur -1 signalant la fin de l'image courante.
La routine @draw_image affiche l'image img% à la position posx% et posy%. Elle lit les donnés définissant le sprite et utilise la procédure @aff_sprite qui affiche un sprite à une position quelconque de l'écran. Leur position d'affichage sur l'écran est calculée en ajoutant la position relative du sprite et sa position d'affichage sur l'écran.

```
PROCEDURE draw_image(img%,posx%,posy%)
  LOCAL n%
  LOCAL sprite%
  LOCAL px%,py%
  '
  FOR n%=1 TO taille_image%(img%)
    sprite%=image%(img%,n%,1)
    px%=image%(img%,n%,2)+posx%
    py%=image%(img%,n%,3)+posy%
    @aff_sprite(sprite%,px%,py%)
  NEXT n%
RETURN
```

La procédure @demo_images est une procédure de démonstration qui permet de visualiser les images existantes. Elle demande un numéro d'image à l'utilisateur et dessine celle-ci. Si l'utilisateur frappe sur la touche [0], le programme s'arrête. Cette procédure n'a pas grande importance. Ce n'est qu'un banc d'essai vous permettant de tester le générateur d'image et l'utilisation de la fonction @draw_image.

Amélioration de DRAWIMG

Le programme DRAWIMG peut être grandement amélioré au niveau de la création des images. Au lieu d'écrire des numéros de sprites dans des DATA, il serait préférable d'avoir un éditeur permettant de sélectionner et de positionner les sprites à la souris. L'auteur de ces lignes a réalisé il y a quelques années un programme de ce type en *TURBO C*. C'est parfaitement réalisable en *GFA Basic* en moins de 1000 lignes.
Les informations définissant les sprites peuvent être stockées dans un fichier disque. Cela permet d'économiser de la place mémoire si les images sont dessinées avec de nombreux sprites.

Etant donné que les positions des sprites sont définis de manière relative à la position d'affichage de l'image, les variables stockant ces positions d'affichages peuvent être du type byte (symbole |). Cela permet d'économiser 3 octets par variables, donc 6 octets par sprites. Le seul inconvénient de cette technique, c'est qu'un sprite ne peut être affiché à plus de 255 pixels du bord haut gauche de l'image. C'est un défaut plutôt mineur!
Si le nombre de sprites est inférieur à 255, on peut aussi mémoriser les numéros de sprites dans des variables de type |. Cela permet d'économiser encore 3 octets par sprite. La taille mémoire nécessaire pour mémoriser un sprite est de 3 octets, alors que DRAWIMG utilise 12 octets pour faire la même chose. Si le nombre de sprites est compris entre 255 et 65535, on peut utiliser une variable de type & (2 octets) pour mémoriser un numéro de sprite. La place mémoire prise par la définition d'un sprite est alors de 4 octets.
Le codage des images peut être mémorisé dans une zone binaire et non dans un tableau. Cela permet d'utiliser juste la quantité de mémoire nécessaire à une image. La plupart des programmeurs de jeux utilisent des données stockées directement en mémoire et non dans des tableaux numériques.
Le programme DRAWIMG stocke un fichier graphique dans un seul tableau alphanumérique, mais rien ne vous empêche de charger plusieurs blocs de sprites dans différents tableaux. Vous pourrez avoir par exemple un tableau contenant des éléments de visages, un tableau contenant des éléments de voitures, et un tableau contenant des éléments d'animaux.

Performances

Cette technique permet de dessiner des images avec très peu de mémoire.
Etant donné que le codage d'une image ne contient que des références de sprites, la taille de l'image n'a pas de rapport avec son occupation mémoire. Une image de grande dimension dessinée avec peu de sprites peut n'occuper que quelques dizaines d'octets, alors qu'une petite image dessinée avec beaucoup de sprites peut occuper une centaine d'octets.
Si l'on ne considère pas la taille mémoire prise par les éléments graphiques de base, il est possible de coder un nombre gigantesque d'images sur une disquette ou en mémoire. En considérant qu'une image moyenne est fabriqué avec 30 sprites (ce qui est déja beaucoup), le codage de cette image occupe 121 octets (4x30 octets pour les sprites et 1 octet pour le nombre de sprites). On peut mettre 6093 images de 30 sprites sur une disquette double face. Cet exemple est théorique, car les images ne seront pas toutes dessinés avec 30 éléments, mais il met en évidence le gain de mémoire permis par cette technique.

Applications

La technique consistant à dessiner des images par assemblage (ou collage) de sprites peut être utilisés dans de nombreux cas de figures. Voici quelques exemples: dessin d'animaux fantastiques, fabrication de visages, habillements de personnages, dessin de ruelles médiévales. Vous pouvez vous amuser à dessiner des animaux fantastiques à partir d'éléments simples (cornes, faces bestiales, têtes de démons, corps hideux, jambes de cheval, etc.) Un éditeur de créatures fantastiques fonctionnant en haute résolution et réalisé en 3 jours a eu un grand succès pendant la fête d'une école primaire.
On peut dessiner des visages en prenant des formes de visages et en y rajoutant des éléments de bases. Ces éléments sont: cheveux, yeux, nez, bouches, mentons. On peut aussi y rajouter des éléments divers comme des bandeaux, des lunettes, des monocles, quelques cicatrices, etc. Avec une cinquantaine d'éléments graphiques occupant environ 60 Ko, on peut créer plusieurs centaines de visages différents. On jouant sur certains éléments comme les yeux ou la bouche, il est même possible de faire varier l'expression d'un visage (amusement, colère, peur, etc..).
Avec une quarantaine de sprites représentant des murs, des toits, des cheminées, des portes et des fenêtre, il est possible de dessiner un grand nombre de maisons différentes, et même des rues.
Pour la petite histoire, tous les éléments graphiques du jeu de rôle *DONJON MASTER* (murs, créatures, portes, écritures sur les murs, etc..) sont dessinés entièrement avec des sprites.

Pour finir

Les programmes de cet article sont disponibles en téléchargement sur le 3615 ATARI. Ils sont livrés dans un fichier binaire compacté autodécompactable et ne coûtent que quelques francs de téléchargement.

TELECHARGEZ DES CENTAINES DE LOGICIELS

SUR LE

3615 ATARI

INSTRUCTION RC_COPY
Copie rapide de blocs

L'instruction RC_COPY est une instruction graphique du GFA Basic 3.0 permettant de réaliser des affichages rapides et des effets spéciaux graphiques. La voilà disséquée pour vous.

Copie de blocs

RC_COPY permet de copier des blocs de pixels d'un écran à un autre. Les écrans d'origine et de destination peuvent être les mêmes. Elle fonctionne dans les 3 résolutions graphiques du ST et est environ 7 fois plus rapide sur une machine équipée d'un blitter que sur un STF de base. Sa syntaxe est la suivante:

```
RC_COPY ecran1, x1, y1, tx, ty TO ecran2, x2, y2

ecran1:Adresse de l'écran d'origine
x1:Position X du bloc de pixel
y1:Position Y du bloc de pixel
tx:Largeur du bloc de pixel
ty:Hauteur du bloc de pixel
ecran2:Adresse de l'écran de destination
x2:Position X de destination
y2:Position Y de destination
```

Création d'un écran en mémoire

Un écran est une zone mémoire de 32000 octets. Cette zone mémoire peut être créée en déclarant une variable alphanumérique de 32000 octets.

```
ecran$=SPACE$(32000) adr_ecr%=VARPTR(ecran$)
```

L'adresse des variables alphanumériques peut varier à la suite d'une restructuration mémoire (garbage collection). Il faut donc recalculer fréquemment l'adresse du tampon mémoire. Pour effacer l'écran, il suffira de mettre une chaîne vide dans ecran$, ou d'effacer la variable ecran$ avec l'instruction ERASE.
On peut aussi réserver de la mémoire grâce à l'instruction MALLOC. La zone mémoire ainsi déclarée sera fixe. En général, les allocations de mémoire sont réalisées au début des programmes. L'instruction MFREE permet de désallouer une zone mémoire créée par MALLOC.

```
adr_ecr%=MALLOC(32000) VOID MFREE(adr_ecr%)
```

Remarque: l'instruction MALLOC renvoie la valeur 0 si la réservation mémoire n'a pu avoir lieu (manque de mémoire disponible).

Copie d'écran

On peut copier un écran sur un autre grâce à BMOVE, une instruction de copie de mémoire. L'exemple suivant copie 32 000 octets de l'adresse ecran1% à l'adresse ecran2%. Les 32 000 octets se trouvant l'adresse ecran2% sont effacés par l'opération de copie.

```
BMOVE ecran1%, ecran2%, 32000
```

L'adresse de l'écran physique (celui qui est affiché à l'écran) peut être determinée avec la fonction XBIOS(2). On peut donc copier le contenu de l'écran physique en mémoire et vice-versa.

```
BMOVE XBIOS(2), adr_ecr%, 32000 BMOVE adr_ecr%,
XBIOS(2), 32000
```

Fabrication d'un écran aléatoire

Pour tester des routines graphiques, il faut avoir une image de test. Celle-ci peut être chargée à partir d'un disque, ou fabriquée sur place. Nous vous proposont ici une petite routine fabriquant une image aléatoire. Elle s'adapte automatiquement à la résolution de l'écran, grâce à la fonction XBIOS(4) qui indique la résolution courante du système.

```
PROCEDURE gen_ecran
  LOCAL l%,c$,nb_car%
  '
  SELECT XBIOS(4)
  CASE 0
    nb_car%=40
  CASE 1
    nb_car%=80
  CASE 2
    nb_car%=80
```

```
  ENDSELECT$
  '
  FOR l%=1 TO 24
    c$=CHR$(RANDOM(26)+65)
    PRINT STRING$(nb_car%,c$);
  NEXT l%
  c$=CHR$(RANDOM(26)+65)
  PRINT STRING$(nb_car%-1,c$);
RETURN
```

La procédure GEN_ECRAN affiche 25 lignes de caractères. Le dernier caractère de la dernière ligne n'est pas affiché afin de ne pas déclencher un scrolling d'écran qui effacerait la première ligne.

Affichage d'un morceau d'écran

Le programme suivant charge une image aléatoire dans un écran virtuel à l'adresse adr_ecr%. Un morceau de cette image est ensuite recopié sur l'écran physique avec l'instruction RC_COPY.

```
RESERVE 100000
adr_ecr%=MALLOC(32000)
@GEN_ECRAN
BMOVE XBIOS(2), adr_ecr%, 32000
CLS
VOID INP(2)
RC_COPY adr_ecr%,0,0,100,100 TO XBIOS(2),0,0
VOID INP(2)
VOID MFREE(adr_ecr%)
END
```

L'écran virtuel est créé avec l'instruction MALLOC. La mémoire prise est libérée en fin de programme par l'instruction MFREE Le programme fonctionne dans les 3 résolutions.

Copie de blocs de pixels directement sur l'écran

L'instruction RC_COPY peut être utilisée pour copier des blocs de pixels directement sur l'écran physique. Ces blocs ne doivent évidemment pas coïncider.

```
RC_COPY XBIOS(2),0,0,32,32 TO XBIOS(2),100,100
```

C'est plus rapide et cela prend moins de mémoire que la combinaison classique des instructions GET et PUT.

```
GET 0,0,31,31, IMG$ PUT 100,100,IMG$
```

Affichage d'une image ligne à ligne

L'instruction RC_COPY permet d'afficher une image ligne à ligne. Pour des raisons de simplicité, nous allons commencer par étudier l'affichage ligne à ligne uniquement en basse résolution. L'écran est divisé en 20 lignes. Etant donné qu'il y a 200 pixels de haut, chaque ligne a une hauteur de 10 pixels (200/20=10). Les lignes sont numérotées de 1 à 20.

```
PROCEDURE aff_ligne (ligne%)
  LOCAL yl%
  '
  yl% = (ligne%-1)*10
  RC_COPY adr_ecr%,0,yl%,320,10 TO XBIOS(2),0,yl%
RETURN
```

Pour afficher une image à partir du haut de l'écran, il faut afficher les lignes dans un ordre croissant. La procédure AFF_HAUT affiche une image à partir du haut. Cet exemple est incomplet. Il manque l'initialisation de l'écran virtuel (avec SPACE$ ou MALLOC), et la fabrication de l'image de test (avec GEN_ECRAN).

```
PROCEDURE aff_haut
  FOR l%=1 TO 20
    @aff_ligne (l%)
    PAUSE 2
  NEXT l%
RETURN
```

Pour afficher une image à partir du bas, il faut afficher les lignes dans un ordre décroissant. La procédure AFF_BAS affiche une image à partir du bas de l'écran.

```
PROCEDURE aff_bas
  FOR l%=20 TO 1 STEP -1
    @aff_ligne(l%)
    PAUSE 2
  NEXT l%
RETURN
```

L'instruction PAUSE 2 sert à fixer un délai de temporisation entre l'affichage des lignes. En augmentant ou en diminuant le paramètre de temporisation, vous ferez varier la vitesse d'affichage de l'image.

Affichage ligne à ligne multi-résolution

Programmer une routine multi-résolution pose toujours quelques problèmes. Il faut déterminer les paramétres qui dépendent de la résolution et écrire une routine d'initialisation.
Dans le cas présent, les paramétres dépendant de la résolution sont la largeur d'une ligne (variable xl%), et la hauteur d'une ligne (variable yl%). La routine DEF_LIGNE calcule leur valeur en fonction du mode graphique. Elle doit impérativement être appelée au début du programme sous peine de mauvais fonctionnement.

```
PROCEDURE def_ligne
  SELECT XBIOS(4)
  CASE 0
    xl%=320
    yl%=10
  CASE 1
    xl%=640
    yl%=10
  CASE 2
    xl%=640
    yl%=20
```

```
  ENDSELECT
RETURN

PROCEDURE aff_ligne(ligne%)
  LOCAL py%
  '
  py%=(ligne%-1)*yl%
  RC_COPY adr_ecr%,0,py%,xl%,yl% TO XBIOS(2),0,py%
RETURN
```

Les procédures AFF_HAUT, et AFF_BAS du paragraphe précédent fonctionnent parfaitement avec la nouvelle procédure AFF_LIGNE Il est amusant de voir la même routine graphique fonctionner dans les 3 résolutions du ST. En anticipant un petit peu, on pourrait imaginer des routines graphiques fonctionnant dans tous les modes graphiques ST et TT.
La plupart des programmes professionnels sur IBM PC et compatibles utilisent un système graphique similaire à celui que nous venons d'étudier. Il existe tellement de cartes graphiques différentes sur IBM PC que c'est une obligation pour les programmeurs.

Affichage d'une ligne sur deux

On peut obtenir un bel effet graphique en affichant les lignes impaires, puis les lignes paires. L'exemple suivant fait apparaître une image à partir du haut, mais le principe fonctionne aussi pour un affichage à partir du bas.

```
PROCEDURE aff_haut2
  FOR l%=1 TO 19 STEP 2
    @aff_ligne (l%)
    PAUSE 2
  NEXT l%
  FOR l%= 2 TO 20 STEP 2
    @aff_ligne (l%)
    PAUSE 2
  NEXT l%
RETURN
```

Cette technique d'affichage peut être utilisée avec d'autres écarts de ligne. Afficher une ligne sur 4 donne un très bel effet.

Affichage des lignes à partir des bords de l'écran

Avec cette technique, l'image apparaît à partir du haut et du bas. Les lignes ligne1% et ligne2% sont affichés quasi-simultanément par la routine. L'instruction PAUSE 1 impose un petit délai d'attente entre les différentes étapes d'affichage.

```
ligne1%=1
ligne2%=20
FOR i%= 1 TO 10
  @aff_ligne(ligne1%)
  @aff_ligne(ligne2%)
  PAUSE 1
  INC ligne%
  DEC ligne2%
NEXT i%
```

En inversant le sens d'affichage des lignes, vous pouvez afficher une image à partir du milieu de l'écran.

```
ligne1%=10
ligne2%=11
FOR i%=1 TO 10
  @aff_ligne(ligne1%)
  @aff_ligne(ligne2%)
  PAUSE 1
  DEC ligne1%
  INC ligne2%
NEXT i%
```

Affichage d'une image colonne par colonne

De la même manière que l'on affiche une image ligne à ligne, on peut afficher une image colonne par colonne. Le principe est le même, seul le sens d'affichage diffère.
L'écran est divisé en 20 colonnes. En basse résolution, les colonnes ont une largeur de 16 pixels (320/20) et une hauteur de 200 pixels. En moyenne résolution, les colonnes ont une largeur de 32 pixels (640/20) et une hauteur de 200 pixels. En haute résolution, les colonnes ont une largeur de 32 pixels (640/20) et une hauteur de 400 pixels.

```
PROCEDURE def_colonne
  SELECT XBIOS(4)
  CASE 0
    xc%=16
    yc%=200
  CASE 1
    xc%=32
    yc%=200
  CASE 2
    xc%=32
    yc%=400
  ENDSELECT
RETURN
```

La procédure DEF_COLONNE initialise les variables xc% et yc% en fonction de la résolution du système. Elle doit être appelée avant d'utiliser la procédure AFF_COLONNE Le plus simple est de l'utiliser au début du programme.

```
PROCEDURE aff_colonne(colonne%)
  LOCAL px%
  '
  px%=(colonne%-1)*xc%
  RC_COPY adr_ecr%,px%,0,xc%,yc% TO XBIOS(2),px%,0
RETURN
```

La procédure AFF_COLONNE affiche la colonne colonne% en tenant compte de la résolution. Les colonnes sont numérotées de 1 à 20. En

affichant les colonnes dans un ordre croissant, l'image s'affiche à partir de la gauche.

```
PROCEDURE aff_gauche
  LOCAL c%
  '
  FOR c%=1 TO 20
    @aff_colonne(c%)
    PAUSE 2
  NEXT c%
RETURN
```

En affichant les colonnes dans un ordre décroissant à partir de 20, l'image s'affiche à partir de la droite de l'écran.

```
PROCEDURE aff_droite
  LOCAL c%
  '
  FOR c%=20 TO 1 STEP -1
    @aff_colonne(c%)
    PAUSE 2
  NEXT c%
RETURN
```

Affichage d'une colonne sur deux

De la même manière que l'on peut afficher une ligne sur deux, on peut afficher une colonne sur deux. L'exemple suivant affiche les colonnes impaires, puis les colonnes paires.

```
PROCEDURE aff_gauche2
  LOCAL c%
  '
  FOR c%=1 TO 19 STEP 2
    @aff_colonne(c%)
    PAUSE 2
  NEXT c%
  FOR c%=2 TO 20 STEP 2
    @aff_colonne(c%)
    PAUSE 2
  NEXT c%
RETURN
```

Vous pouvez aussi afficher une image colonne par colonne à partir des bords ou du centre de l'écran. Inspirez-vous des techniques d'affichages ligne à ligne.

Découpage d'une image en petits rectangles

Au lieu de découper l'écran en lignes et colonnes, on peut le découper en petits rectangles. Cela permet de créer de nouveaux effets graphiques. Afin de maintenir une compatibilité avec les programmes précédents, l'écran est découpé en 20x20 blocs (soit 400 blocs).
En basse résolution, un bloc graphique fait 16x10 pixels. En moyenne résolution, un bloc graphique fait 32x10 pixels. En haute résolution, un bloc graphique fait 32x20 pixels. La procédure DEF_BLOC initialise les variables xb% et yb% en fonction de la résolution du système. Elle doit être exécutée au début des programmes.

```
PROCEDURE def_bloc
  SELECT XBIOS(4)
  CASE 0
    xb%=16
    yb%=10
  CASE 1
    xb%=32
    yb%=10
  CASE 2
    xb%=32
    yb%=20
  ENDSELECT
RETURN
```

La procédure AFF_BLOC copie un bloc graphique de l'image stockée à l'adresse adr_ecr% sur l'écran physique. Les lignes et les colonnes définissant la position des blocs sont numérotées de 1 à 20.

```
PROCEDURE aff_bloc(ligne%,colonne%)
  LOCAL px%, py%
  '
  px%=(colonne%-1)*xb%
  py%=(ligne%-1)*yb%
  RC_COPY adr_ecr%,px%,py%,xb%,yb% TO XBIOS(2),px%,py%
RETURN
```

La routine AFF_BLOC peut être utilisée pour réaliser des affichages sophistiqués comme l'affichage aléatoire, ou l'affichage en diagonale.

Affichage aléatoire

L'ordre d'affichage des blocs est déterminé au hasard. Le programme doit garder la trace des blocs déjà affichés afin de ne pas afficher deux fois le même. Le tableau bloc%() contient la liste des blocs déja affichés à l'écran. Au début, les cases de ce tableau contiennent toutes 0. Lorsque le programme affiche un bloc sur l'écran, il met la valeur 1 dans la case correspondant au bloc affiché. La routine de tirage aléatoire sait qu'un bloc n'est pas encore affiché si la case équivalente de bloc%() contient la valeur 0.

```
DIM bloc%(20,20)
ARRAYFILL bloc%(),0
FOR i%=1 TO 400
  REPEAT
    c%=RAND(20)+1
    l%=RAND(20)+1
  UNTIL bloc%(l%,c%)=0
  @aff_bloc(l%,c%)
  bloc%(l%,c%)=1
NEXT i%
```

L'instruction ARRAYFILL bloc%(),0 met la valeur 0 dans toutes les cases du tableau bloc%(). L'instruction RAND(n) fournit un nombre aléatoire compris entre 0 et n-1. La formule RAND(n)+1 donne donc

un nombre aléatoire compris entre 1 et n. Associé à la procédure DEF_BLOC, ce programme permet de réaliser une routine d'affichage aléatoire fonctionnant dans les 3 résolutions du ST.

Amélioration de l'affichage aléatoire

Si vous avez exécuté la routine d'affichage aléatoire, vous avez dû remarquer que l'affichage des derniers blocs est un peu lent. Le programme peine un peu pour trouver les 5 ou 6 derniers blocs non affichés parmi les 400 blocs possibles. Ceci dis, le *GFA Basic* est tellement rapide qu'il met moins d'une seconde à trouver le dernier bloc (en version interprétée).
Pour éviter ce petit ralentissement, il faut arrêter le calcul aléatoire des blocs avant la fin, et afficher rapidement les derniers blocs restant. On peut, par exemple, n'afficher que 395 blocs et utiliser une routine supplémentaire utilisant le tableau bloc%() pour connaître les blocs non affichés.

```
nb_blocs%=395
FOR i%=1 TO nb_blocs%
 ' ..AFFICHAGE ALEATOIRE
NEXT i%
'
FOR c%=1 TO 20
   FOR l%=1 TO 20
     IF bloc%(l%,c%)=0
        @aff_bloc(l%,c%)
        bloc%(l%,c%)=1
        PAUSE 1
     ENDIF
   NEXT l%
NEXT c%
```

L'instruction PAUSE 1 évite que les derniers blocs ne soient affichés trop rapidement. Vous pouvez la retirer si vous trouvez qu'elle ne sert à rien. Vous pouvez faire varier le nombre de blocs affichés aléatoirement, en modifiant la variable nb_blocs% (elle contient 395 dans le programme d'exemples).

Affichage d'une image en diagonale

Les routines d'affichage en diagonale donnent toujours un résultat agréable à regarder, mais elles sont un peu plus difficiles à programmer que les autres.
Nous avons vu que l'écran est divisé en 400 blocs. Ces blocs sont organisés en 20 lignes et en 20 colonnes. C'est une situation idéale qui simplifie considérablement la programmation. Les diagonales sont à 45 degrés. Pour tracer une diagonale, il faut partir d'un bloc graphique, puis décrémenter régulièrement la position du bloc courant avant d'arriver au bord de l'écran. La diagonale commençant au bloc (10,10) est constituée des blocs (10,10), (9,9), (8,8), .., (1,1).

```
PROCEDURE diagonale(l%,c%)
   REPEAT
     @aff_bloc(l%,c%)
     DEC c%
     DEC l%
   UNTIL (c%<1) OR (l%<1)
RETURN
```

La procédure DIAGONALE cesse le dessin lorsque le bloc à afficher sort des limites de l'écran, c'est-à-dire lorsque le numéro de ligne devient inférieur à 1, ou lorsque le numéro de colonne devient inférieur à 1. Cette technique trace une diagonale en direction du coin haut gauche de l'écran. On peut tracer des diagonales suivant d'autres directions (vers le coin haut droit, vers le coin bas gauche, ou encore vers le coin bas droit de l'écran).
A titre d'exemple, la routine DIAGO_BIS trace une diagonale en direction du coin haut droit de l'écran. Cette nouvelle routine s'arrête lorsque le bloc à afficher est hors de l'écran, c'est-à-dire lorsque le numéro de ligne est inférieur à 1, ou lorsque le numéro de colonne est supérieur à 20.

```
PROCEDURE diago_bis(l%,c%)
   REPEAT
     @aff_bloc(l%,c%)
     INC c%
     DEC l%
   UNTIL (c%>20) OR (l%<1)
RETURN
```

Affichage d'une diagonale particulière

Il y a 39 diagonales sur un écran. La diagonale 1 est située en bas et à gauche de l'écran. La diagonale 39 est située en haut et à droite de l'écran. Pour réaliser diverses routines d'affichage, il faut pouvoir afficher sélectivement n'importe quelle diagonale.
La routine AFF_DIAGONAL est capable d'afficher une diagonale de l'écran à partir de son numéro.

```
PROCEDURE aff_diagonal(num%)
   LOCAL ligne%
   LOCAL colonne%
   '
   IF num%<=20
     ligne%=20
     colonne%=num%
   ELSE
     ligne%=40-num%
     colonne%=20
   ENDIF
   @diagonale(ligne%,colonne%)
RETURN
```

Cette routine reconstitue la position du bloc de départ à partir du numéro de diagonale.

Affichage en diagonale de la gauche vers la droite

La procédure IMG_DIAGONALE1 affiche une image en diagonale en partant de la gauche. Elle affiche les différentes diagonales les unes après les autres. La routine est très simple, car le travail a été décomposé niveau par niveau.

```
PROCEDURE img_diagonale1
   LOCAL d%
   '
   FOR d%=1 TO 39
     @aff_diagonal(d%)
     PAUSE 1
   NEXT d%
RETURN
```

Affichage en diagonale de la droite vers la gauche

La procédure IMG_DIAGONALE2 affiche une image en diagonale en partant de la droite. La diagonale 39 est affichée en premier.

```
PROCEDURE img_diagonale2
   LOCAL d%
   '
   FOR d%=39 TO 1 STEP -1
     @aff_diagonal(d%)
     PAUSE 1
   NEXT d%
RETURN
```

Affichage d'une diagonale sur deux

En affichant une diagonale sur deux, on obtient un effet graphique assez joli. La procédure AFF_2DIAGONALE affiche une image à partir de la gauche. Elle a été écrite en modifiant la procédure IMG_DIAGONALE1. Vous pourrez modifier de la même manière la procédure IMG_DIAGONALE2 pour obtenir un affichage à partir de la droite.

```
PROCEDURE aff_2diagonale
   LOCAL d%
   '
   FOR d%=1 TO 39 STEP 2
     @aff_diagonal(d%)
     PAUSE 1
   NEXT d%
   '
   FOR d%= 2 TO 38 STEP 2
     @aff_diagonal(d%)
     PAUSE 1
   NEXT d%
RETURN
```

Bibliothèque d'images

Vous pouvez utiliser RC_COPY pour créer une petite bibliothèque d'images. Le principe de base est de dessiner des images avec un logiciel de dessin type *Degas Elite*, puis de charger en mémoire l'image écran générée par le logiciel graphique. Il suffira ensuite d'utiliser RC_COPY pour recopier les images sur l'écran physique.
La procédure LOAD_IMAGE charge l'image nom$ à l'adresse adr_ecr%. Si cette variable ne contient pas l'adresse d'une zone mémoire de 32000 octets préparée avec MALLOC ou VARPTR, vous risquez de provoquer un superbe plantage de la machine.

```
PROCEDURE load_images(nom$)
   OPEN "I",#1,nom$
   SEEK #1,34
   BGET #1,adr_ecr%,32000
   CLOSE #1
RETURN
```

La position et la taille des images doivent être mémorisées dans un tableau numérique. Ces données doivent être chargées dans le tableau au début du programme. Ce peut être fait avec des instructions DATA, ou en lisant le contenu d'un fichier.

```
DIM image%(100,4)
 ' image%(n,1) = position x
 ' image%(n,2) = position y
 ' image%(n,3) = largeur (tx)
 ' image%(n,4) = hauteur (ty)
```

La procédure AFF_IMAGE affiche à la position (x%,y%) de l'écran, l'image numéro n%. L'image *Degas Elite* doit être chargée en mémoire à l'adresse adr_ecr%.

```
PROCEDURE aff_image(x%,y%,n%)
  LOCAL px%, py%
  LOCAL tx%, ty%
  '
  px% = image%(n%,1)
  py% = image%(n%,2)
  tx% = image%(n%,3)
  ty% = image%(n%,4)
  RC_COPY adr_ecr%,px%,py%,tx%,ty% TO XBIOS(4),x%,y%
RETURN
```

Extension de la bibliothèque d'images

Le système de gestion de bibliothèque d'images que nous venons de voir ne peut pas en gérer beaucoup. On est vite limité par la taille maximale d'une image. On peut avoir davantage d'images en travaillant sur plusieurs fichiers *Degas Elite*. LOAD_IMAGE doit être modifiée pour charger une image *Degas* à une adresse particulière.

```
PROCEDURE load_images(nom$,adr_img%)
   OPEN "I",#1,nom$
   SEEK #1,34
   BGET #1,adr_img%,32000
   CLOSE #1
RETURN
```

Le tableau image%() doit contenir une case supplémentaire pour contenir le numéro d'écran de l'image n%. L'exemple déclare un tableau de 100 images, mais il peut être beaucoup plus grand avec ce système d'écrans multiples (1 000 images par exemple).

```
DIM image%(100,5)
```

Les nouveaux écrans doivent être créés au début du programme. Les adresses peuvent être mémorisées dans un tableau. La réservation de

zone mémoire se fait avec l'instruction MALLOC. L'exemple suivant réserve de la place mémoire pour 4 images.

```
DIM adr_ecran%(4)
FOR i%=1 TO 4
   adr_ecran%(i%)=MALLOC(32000)
NEXT i%
```

Attention: 4 écrans de 32000 octets représentent 128 000 octets. Il est possible que le système manque de mémoire (surtout sur 520) si la mémoire est déja bien encombrée (programme important, tableaux de grande taille, nombreux accessoires, programmes résidents, etc.) Pour plus de sécurité, il faut contrôler la bonne exécution des réservations mémoire.

```
FOR i%=1 TO 4
   adr%=MALLOC(32000)
   IF adr%=0
      PRINT "Erreur mémoire"
      END
   ENDIF
   adr_ecran%(i%)=adr%
NEXT i%
```

Une fois le programme terminé, la mémoire réservée doit être libérée. Cette opération se fait avec l'instruction MFREE

```
FOR i%=1 TO 4
   VOID MFREE(adr_ecran%(i%))
NEXT i%
```

La procédure AFF_IMAGE s'utilise toujours de la même manière pour le programmeur. Par contre, il y a une différence importante au niveau interne: la routine détermine l'adresse de l'image DEGAS ELITE avant d'appeler l'instruction RC_COPY.

```
PROCEDURE aff_image(x%,y%,n%)
  LOCAL px%, py%
  LOCAL tx%, ty%
  LOCAL num_image%
  LOCAL adr_img%
  '
  px% = image%(n%,1)
  py% = image%(n%,2)
  tx% = image%(n%,3)
  ty% = image%(n%,4)
  num_img% = image%(n%,5)
  adr_img%=adr_ecran%(num_img%)
  RC_COPY adr_img%,px%,py%,tx%,ty% TO XBIOS(4),x%,y%
RETURN
```

Une image est définie par 5 paramétres: position X, position Y, largeur (tx), hauteur (ty) et numéro de l'image *Degas* contenant cette image.
Les positions des images peuvent être définies en mémoire dans des lignes de DATA.
La valeur -1 sert d'indicateur de fin de données.

```
images:
'
DATA px,py,tx,ty,num_image
DATA 10,10,32,32,1
DATA 80,10,12,17,1
DATA 10,10,32,32,2
DATA -1
```

La routine INIT_IMAGES lit les informations stockées dans les lignes de DATA et les stocke dans le tableau image%(). La variable nb_images% contient le nombre d'images lues.

```
PROCEDURE init_images
   LOCAL px%,py%,tx%,ty%
   LOCAL num_img%
   LOCAL n%
   '
   n%=0
   RESTORE images
   DO
     READ px%
     EXIT IF px%=-1
     INC n%
     READ py%,tx%,ty%
     READ num_img%
     image%(n%,1)=px%
     image%(n%,2)=py%
     image%(n%,3)=tx%
     image%(n%,4)=ty%
     image%(n%,5)=num_img%
   LOOP
   nb_images%=n%
RETURN
```

Le chargement des images *Degas* doit se faire au début du programme, mais aprés la réservation mémoire. L'exemple charge une bibliothèque d'images utilisée par un mini-jeu de rôle.

```
@load_ecran("OBJETS.PI1",  adr_ecran%(1))
@load_ecran("ARMES.PI1",   adr_ecran%(2))
@load_ecran("MONSTRE.PI1", adr_ecran%(3))
@load_ecran("VILLE.PI1",   adr_ecran%(4))
```

Scrolling d'une zone de l'écran

Une routine de scrolling permet de simuler le déplacement d'une fenêtre virtuelle sur une image de grande dimension. Au point de vue technique, il s'agit d'un affichage rapide de bloc d'images. L'instruction RC_COPY permet de réaliser cela facilement.
Il faut travailler très rapidement pour obtenir un scrolling de grande dimension parfaitement fluide. C'est impossible à réaliser avec RC_COPY sur un STF, car la routine est trop lente. On ne peut obtenir que des scrollings de toutes petites dimensions. L'exemple présenté dans cet article ne fonctionne correctement que sur un STE, ou un MEGA ST avec blitter actif.
La routine AFF_ZONE affiche au milieu de l'écran un bloc graphique de 192x64 pixels. Ce bloc graphique provient d'un écran virtuel si-

tué à l'adresse adr_ecr%. La routine prend le bloc situé à la position (px%,py%) de l'écran virtuel et le copie à la position (64,68) de l'écran visible. Afin d'éviter des bavures vidéo, AFF_ZONE attend que la barre synchronisation vidéo ne soit pas sur l'écran pour réaliser le transfert de pixels (grâce à l'instruction VSYNC).

```
PROCEDURE aff_zone(px%,py%)
  VSYNC
  RC_COPY adr_ecr%,px%,py%,192,64 TO XBIOS(2),64,68
RETURN
```

L'exemple suivant effectue un scrolling vertical (du haut vers le bas) de 101 pixels. Il est réalisé en recopiant 101 fois une partie de l'écran adr_ecr% sur l'écran en changeant à chaque fois la position du bloc graphique d'un pixel.
L'instruction BOX affiche un rectangle graphique autour de la zone de scrolling. Ce rectangle n'a pas de réélle utilité pour le scrolling. Il est juste là pour des raisons esthétiques. L'esthétisme est une chose importante en informatique, quoi qu'en pensent beaucoup de programmeurs.

```
COLOR 2 BOX 63,67,256,132
FOR py%=0 TO 100
   @aff_zone(100,py%)
NEXT py%
```

Pour réaliser un scrolling du bas vers le bas, il faut afficher les blocs graphiques dans le sens inverse.

```
FOR py%=100 TO 0 STEP -1
   @aff_zone(100,py%)
NEXT py%
```

La vitesse de scrolling dépend du pas d'incrémentation des pixels. Le pas d'incrémentation est le nombre ajouté à une position d'affichage pour obtenir une nouvelle position. Il était de 1 dans les exemples précédents. Vous pouvez augmenter la vitesse de scrolling en augmentant le pas d'incrémentation. L'exemple suivant utilise un pas d'incrémentation de 2, afin de doubler la vitesse de scrolling.

```
FOR py%=1 TO 100 STEP 2
   @aff_zone(100,py%)
NEXT py%
```

Vous pouvez aussi vous amuser à réaliser un scrolling double en modifiant simultanément les positions X et Y du bloc graphique.

```
px%=0
py%=0
FOR i%= 0 TO 199
   @aff_zone(px%,py%)
   INC px%
   INC py%
NEXT i%
```

Encore plus fort: faites un scrolling double avec des pas d'incrémentation différents pour les deux axes.

```
px%=0
py%=0
FOR i%= 0 TO 150
   @aff_zone(px%,py%)
   ADD px%,2
   INC py%
NEXT i%
```

Que vous utilisiez un blitter ou non, il est impossible de réaliser un scrolling plein écran suffisamment rapide avec l'instruction RC_COPY. Cela ne vous en empêche pas de l'utiliser pour réaliser divers petits effets graphiques. Le nom d'un logiciel accompagné d'un petit dessin peut apparaître en scrollant dans une petite fenêtre. Un long texte peut être visualisé par petits bouts dans une fenêtre. Une série d'icônes peuvent défiler sur l'écran. Certains jeux d'aventure utilisent une routine de scrollings pour faire défiler doucement leurs divers textes.

Rafraîchissement de fenêtres graphiques

Lorsque vous modifiez les caractéristiques d'une fenêtre graphique GEM (taille, position, déplacement dans la fenêtre avec les ascenseurs, etc.), la fenêtre est redessinée. Cette opération (redraw dans la terminologie GEM, redessin, ou encore rafraîchissement) peut être réalisée par l'instruction RC_COPY.
Une fenêtre graphique est définie par ces paramètres écran (position de la fenêtre sur l'écran, taille de la fenêtre visible), et par un paramétre interne (position de la fenêtre visible par rapport à la fenêtre virtuelle). La fenêtre virtuelle est stockée quelque part en mémoire.

fenetre : Adresse mémoire de la fenêtre virtuelle;
xpos : Position X de la fenêtre visible par rapport à la fenêtre virtuelle;
ypos : Position Y de la fenêtre visible par rapport à la fenêtre virtuelle;
tx : Largeur de la fenêtre visible;
ty : Hauteur de la fenêtre visible;
px : Position X de la fenêtre visible sur l'écran;
py : Position Y de la fenêtre visible sur l'écran.

```
RC_COPY fenetre, xpos, ypos, tx, ty TO XBIOS(2), px, py
```

TELECHARGEZ DES CENTAINES DE LOGICIELS SUR LE 3615 ATARI

LES OPERATEURS GRAPHIQUES LOGIQUES

Manipulez les pixels

Les fonctions graphiques logiques du GFA Basic permettent de réaliser de nombreux effets graphiques (affichage de sprites, vidéo inverse, effets spéciaux, etc.) Découvrons-les ensemble.

Fonctions graphiques de base

Le *GFA Basic* peut stocker en mémoire et afficher des blocs graphiques grâce aux instructions GET et PUT. L'instruction GET permet de stocker un bloc graphique dans une variable alphanumérique, alors que PUT permet d'afficher le contenu graphique de la variable n'importe où sur l'écran.

```
GET x1,y1,x2,y2,img$
PUT x,y,img$
```

L'instruction PUT recopie intégralement sur l'écran les pixels mémorisés dans la variable img$. On peut aussi mélanger ces pixels de l'image avec ceux de l'écran. Ce mélange peut se faire de différentes manières grâce à ce que l'on appelle des opérateurs graphiques logiques.

Les opérateurs graphiques logiques

Les opérateurs graphiques logiques sont des fonctions graphiques qui peuvent faire «fusionner» plusieurs pixels entre eux au niveau des couleurs. Les opérateurs graphiques logiques sont des fonctions qui servent à mélanger deux images entre elles. Le résultat obtenu dépend des images, mais aussi du type d'opérateurs graphiques utilisé. Le mélange se fait pixel par pixel. Pour chaque pixel, la couleur résultante dépend de la couleur initiale, de la nouvelle couleur et de l'opérateur logique. Les opérations entre couleurs de pixels se font au niveau binaire, c'est-à-dire que les opérateurs graphiques logiques travaillent bit à bit. Les opérateurs binaires les plus utilisés sont AND, OR, XOR et NOT.

Opérateur graphique logique AND

Lorsqu'on applique l'opérateur AND à deux bits, le résultat ne vaut 1 que si les 2 bits sont à 1.

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

Lorsque l'opérateur graphique logique AND est appliqué à 2 pixels, le résultat dépend du codage binaire de la couleur des pixels.
Par exemple, en appliquant l'opérateur graphique AND sur un pixel de couleur 13 et un pixel de couleur 6, on obtient un pixel de couleur 4.

```
1101  (13 en décimal)
0110  (06 en décimal)
----   AND
0100  (04 en décimal)
```

Mise en œuvre de l'opérateur logique AND

Pour effectuer une opération graphique logique AND entre deux images, il faut stocker l'une des images dans une variable alphanumérique avec l'instruction GET, puis l'afficher sur l'autre image avec l'instruction PUT en spécifiant le paramètre supplémentaire 1.

```
GET x1,y1,x2,y2,img$
PUT px,py,img$,1
```

Ce petit programme fait une opération logique AND entre l'image définie par les points (x1,y1,x2,y2) et l'image commençant aux coordonnées (px,py). Après son exécution, l'image commençant aux coordonnées (px,py) est la résultante des 2 images de départ.

Intérêt de l'opérateur logique AND

L'opérateur AND est très pratique pour réaliser des affichages en ombre chinoise. Prenons une image basse résolution faite uniquement avec des pixels de couleur 0 et 15. En affichant cette image sur l'écran avec l'opérateur graphique logique AND, on observe un phénomène très important: les pixels de couleurs 0 sont bien visibles sur l'écran, mais ceux de couleur 15 ont disparu.
La couleur 15 se code 1111 en binaire, et la couleur 0 se code 0000

en binaire. Lorsque la routine d'affichage a fait un AND logique entre les pixels de l'écran et ceux de l'image de test, tous les pixels ANDés avec la couleur 0 se sont retrouvés à 0, alors que tous les ANDés avec la couleur 15 n'ont pas été modifiés.

```
n AND 0000 = 0  (0000 = 0 en décimal)
n AND 1111 = n  (1111 = 15 en décimal)
```

On appelle cette technique affichage en ombre chinoise car elle ne permet d'afficher que des images d'une seule couleur.

Programme de démonstration de l'opérateur logique AND

Avant d'étudier le programme de démonstration de l'opérateur graphique logique AND, il faut savoir que le *GFA Basic* utilise sa propre table de couleurs. En effet, après une instruction DEFFILL 2, il tracera des rectangles graphiques de couleur 1, et non 2. Voici la table de conversion couleurs réels/couleurs *GFA Basic*:

```
Couleurs réelles 0 1 2 3 4 5 6 7 8 9  10 11 12 13 14 15
Couleurs GFA Basic 0 2 3 6 4 7 5 8 9 10 11 14 12 15 13 1
```

Les seules couleurs qui ne sont pas modifiées sont les couleurs 0,4 et 12. En fait, il ne s'agit pas d'un bug du *GFA Basic*, mais du système graphique VDI présent dans les ROMs du STE. Il faut utiliser la couleur 1 pour obtenir à l'écran la couleur 15. Attention: les logiciels de dessin graphiques travaillent avec les véritables numéros de couleurs, c'est-à-dire que les pixels tracés avec la couleur 15 sont en couleur 15.

Le programme de démonstration affiche une petite image constituée d'une bande de couleur 0 et d'une bande de couleur 15 (couleur 1 pour le *GFA Basic*). Cette image est stockée dans la variable alphanumérique image$. Elle est affichée à la position (100,10) grâce à l'instruction PUT. Elle est ensuite affichée avec l'opérateur logique AND à la position (200,10).

```
DEFFILL 2,2,2
PBOX 0,0,319,199
'
DEFFILL 0,1
PBOX 10,10,60,60
DEFFILL 1
PBOX 20,20,50,50
DEFFILL 0
PBOX 30,30,40,40
'
GET 10,10,60,60,image$
VOID INP(2)
'
PUT 100,10,image$
VOID INP(2)
'
PUT 200,10,image$,1
VOID INP(2)
'
END
```

En exécutant ce programme, vous pourrez vérifier que les pixels de couleur 0 sont affichés sur l'écran et que les pixels de couleur 15 ont disparu. Cette technique fonctionne aussi en moyenne et haute résolution, mais les numéros de couleurs changent. Etant donné que les couleurs en moyenne résolution sont codés sur 2 bits, il faut travailler avec les couleurs 0 et 3 (0 et 6 en tenant compte de la bug des couleurs VDI). De même pour la haute résolution, il faut travailler avec les couleurs 0 et 1 (0 et 2 pour tenir compte de la bug des couleurs VDI).

Opérateur graphique logique OR

Lorsqu'on applique l'opérateur OR à deux bits, le résultat ne vaut 1 que si au moins l'un des deux bits est à 1.

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

En appliquant l'opérateur graphique OR sur un pixel de couleur 13 et un autre de couleur 6, on obtient un pixel de couleur 15.

```
1101  (13 en décimal)
0110  (06 en décimal)
----   OR
1111  (15 en décimal)
```

Les propriétés importantes de l'opérateur graphique logique OR sont les suivantes:

```
n OR 0000 = n
n OR 1111 = 1111
```

Démonstration de l'opérateur logique OR

L'opérateur graphique logique OR est obtenu de la manière suivante: PUT px,py,image$,7. Le même programme de démonstration que AND donne le résultat suivant: tous les pixels de couleurs 15 sont visibles après l'affichage en mode OR, alors que tous les pixels de couleurs 0 ont disparu. C'est un résultat inverse de celui obtenu avec l'opérateur logique AND.

```
DEFFILL 2,2,2
PBOX 0,0,319,199
'
DEFFILL 0,1
PBOX 10,10,60,60
DEFFILL 1
PBOX 20,20,50,50
DEFFILL 0
PBOX 30,30,40,40
'
GET 10,10,60,60,image$
VOID INP(2)
'
PUT 100,10,image$
```

```
VOID INP(2)
'
PUT 200,10,image$,7
VOID INP(2)
'
END
```

De la même manière que pour l'opérateur AND, on peut utiliser l'opérateur OR en moyenne et haute résolution en changeant le numéro des couleurs.

Opérateur logique XOR

Lorsqu'on applique l'opérateur XOR à deux bits, le résultat ne vaut 1 que si seulement l'un des deux bits est à 1.

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 0
```

En appliquant l'opérateur graphique OR sur un pixel de couleur 13 et un de couleur 6, on obtient un pixel de couleur 11.

```
1101  (13 en décimal)
0110  (06 en décimal)
----   OR
1011  (11 en décimal)
```

L'opérateur graphique logique XOR est réversible, c'est-à-dire que l'on permute une opération logique pour revenir aux données de départ (13 XOR 6 = 11; et 11 XOR 6 = 13).

```
n XOR a = b
b XOR a = n
```

Utilisation de l'opérateur logique XOR

L'intérêt de XOR est de pouvoir afficher un objet sur l'écran, et de revenir à l'image initiale en réaffichant l'objet une seconde fois. Cela peut paraître curieux, mais fonctionne bien. Le programme de démonstration suivant vous convaincra.

```
DEFFILL 2,2,2
PBOX 0,0,319,199
'
DEFFILL 0,1
PBOX 10,10,60,60
DEFFILL 1
PBOX 20,20,50,50
DEFFILL 0
PBOX 30,30,40,40
'
GET 10,10,60,60,image$
VOID INP(2)
'
PUT 100,10,image$
VOID INP(2)
'
' **********************
' * AFFICHAGE MODE XOR *
' **********************
PUT 200,10,image$,6
VOID INP(2)
'
' ************************
' * REAFFICHAGE MODE XOR *
' ************************
PUT 200,10,image$,6
VOID INP(2)
END
```

Les premiers jeux d'arcades sur micro-ordinateur utilisaient le mode graphique XOR pour afficher leurs sprites. Le résultat était plutôt curieux au niveau des couleurs, mais cela fonctionnait rapidement et les utilisateurs de l'époque n'en demandaient pas plus.

Opérateur logique NOT

L'opérateur logique NOT est un peu particulier, puisqu'il ne travaille que sur un seul nombre. Les bits du nombre sont inversés (les 0 deviennent des 1, et les 1 deviennent des 0).

```
NOT 1 = 0
NOT 0 = 1

1101  (13 en décimal)
----   NOT
0010  (2 en décimal)
```

On obtient le mode logique NOT en donnant le paramètre supplémentaire 12 à l'instruction graphique PUT.

```
PUT px,py,image$,12
```

L'opérateur logique NOT sert notamment à inverser les couleurs d'une image. Cela peut être utile pour afficher des zones d'écran en vidéo inverse. Le programme de démonstration suivant inverse une image de test.

```
DEFFILL 2,2,2
PBOX 0,0,319,199
'
DEFFILL 0,1
PBOX 10,10,60,60
DEFFILL 1
PBOX 20,20,50,50
DEFFILL 0
PBOX 30,30,40,40
'
GET 10,10,60,60,image$
VOID INP(2)
'
PUT 100,10,image$
```

```
VOID INP(2)
'
PUT 200,10,image$,12
VOID INP(2)
'
END
```

Les autres opérateurs graphiques logiques

Le GFA possède 16 opérateurs graphiques logiques. Nous n'avons présenté que les 4 principaux. Vous pourrez trouver la description des 12 autres modes à la page 196 du manuel du *GFA Basic 3.0*.

Applications des opérateurs graphiques logiques

Elles sont très nombreuses. Nous allons vous en présenter plusieurs que vous pourrez réutiliser dans vos programmes.

Visualisation d'une icône en vidéo inverse

La vidéo inverse est un moyen pratique de visualiser une option ou une commande en cours d'exécution. Par exemple, le bureau de GEM visualise en vidéo inverse le fichier courant, c'est-à-dire le fichier sur lequel l'utilisateur vient de cliquer. Vous pouvez faire la même chose en utilisant l'opérateur graphique logique NOT.
La vidéo inverse peut être utilisée pour visualiser une action en cours. Supposons que vous ayez programmé un petit soft de gestion utilisant un système de commande à base d'icônes. Lorsque l'utilisateur clique sur une icône, celle-ci passe en vidéo inverse et ne revient en vidéo normale que lorsque le traitement est terminé.
Le programme suivant montre comment ce système peut fonctionner. Lorsqu'une routine de traitement est exécutée, elle efface la souris, saisit le dessin de l'icône dans une variable alphanumérique et réaffiche ce dessin en vidéo inverse grâce à l'opérateur graphique logique NOT. Une fois le traitement achevé, la routine réaffiche l'icône normale avec PUT.

```
PROCEDURE exec_impression
 HIDEM
 GET x1,y1,x2,y2,icone$
 PUT x1,y1,icone$,12
 ......
 ......
 ......
 PUT x1,y1,icone$
 SHOWM
RETURN
```

Cependant, au lieu d'afficher l'icône en vidéo inverse, vous pouvez vous amuser à afficher une petite image visualisant le traitement en cours. Pour une impression, ce peut être une plume et du papier. Pour un tri, un facteur regardant une énorme pile de papier d'un air découragé, etc.

Affichage d'une police de caractères

L'opérateur graphique AND permet de programmer une routine d'affichage de caractères. On peut ainsi créer de nouveaux caractères de n'importe quelle taille et les utiliser dans les programmes en GFA Basic.
Nous avons vu que l'opérateur graphique AND peut afficher des dessins dont seuls les pixels en couleur 0 sont visibles. Si ces pixels visibles forment la matrice d'un dessin, il suffit alors d'afficher le dessin d'un caractère en mode AND pour voir apparaître celui-ci. La matrice du caractère doit être dessiné dans la couleur 0, et les autres pixels doivent être en couleur 15.

```
PUT 100,100,caractere$,1
```

Les différentes matrices doivent êtres stockées dans un tableau alphanumérique. Pour afficher le caractère n, il suffit d'afficher le dessin numéro n.

```
DIM car$(255)
PUT 100,100,car$(n)
```

Les caractères sont toujours affichés avec la couleur 0. Cela peut paraître contraignant, mais il n'en est rien. Comme par hasard, beaucoup de logiciels commercialisés n'affichent que des caractères en couleur 0 (ou en couleur 15).
La meilleure manière de créer vos nouveaux caractères est d'utiliser un logiciel de dessin type *Degas Elite*. Une fois les caractères dessinés, il faut utiliser un programme de saisie générant un fichier binaire.

```
' ***************************
' *     GENERATEUR DE       *
' *  POLICE DE CARACTERES   *
' ***************************
RESERVE 50000
DIM car$(255)
@charge_degas("POLICE.PI1")
@saisie_car
@sauve_police("POLICE.CAR")
END
'
' ***************************
' * POSITION DES CARACTERES *
' ***************************
' DATA px, py, num car
' -1 = fin de DATA
DATA 0,0,65
DATA 8,0,66
DATA 16,0,67
DATA 24,0,68
DATA 32,0,69
DATA -1
'
PROCEDURE saisie_car
 LOCAL px,py,n
 '
 DO
  READ px
  EXIT IF px=-1
  READ py
```

```
  READ n
  GET px,py,px+7,py+7,car$(n)
 LOOP
RETURN
'
PROCEDURE sauve_police(nom$)
 LOCAL i,l
 '
 OPEN "o",#1,nom$
 FOR i=0 TO 255
  l=LEN(car$(i))
  IF l=0
    OUT #1,0
  ELSE
    OUT #1,l
    BPUT #1,VARPTR(car$(i)),l
  ENDIF
 NEXT i
RETURN
```

Le programme suivant charge la police de caractères POLICE.CAR en mémoire, et l'utilise pour afficher un message.

```
' *************************
' *       AFFICHAGE       *
' *  POLICE DE CARACTERES *
' *************************
RESERVE 50000
DIM car$(255)
'
@charge_police("POLICE.CAR")
DEFFILL 2
PBOX 0,0,319,199
@afftext(30,30,"HELLO WORLD")
VOID INP(2)
END
'
PROCEDURE afftext(px,py,m$)
 LOCAL i,xaff,n
 '
 xaff=px
 FOR i=1 TO LEN(m$)
  c$=MID$(m$,i,1)
  n=ASC(c$)
  IF LEN(car$(n))<>0
    PUT xaff,py,car$(n),1
  ENDIF
  ADD xaff,8
 NEXT i
RETURN
'
PROCEDURE charge_police(nom$)
 LOCAL i,l
 '
 OPEN "i",#1,nom$
 FOR i=0 TO 255
  l=INP(#1)
  IF l<>0
    car$(i)=SPACE$(l)
    BGET #1,VARPTR(car$(i)),l
  ENDIF
 NEXT i
RETURN
```

L'affichage d'un message complet se fait caractère par caractère. La routine AFF_MESSAGE affiche un message à la position (px%,py%). Il peut arriver que le programme tente d'afficher un caractère qui n'a pas été redéfini. La routine d'affichage teste donc la taille de car$(n) pour vérifier si le dessin du caractère n existe. Cette routine d'affichage fonctionne avec des caractères d'une largueur de 8 pixels, mais rien ne vous empêche de la modifier pour des caractères plus larges.

Police de caractères de couleur 15

En affichant les caractères avec l'opérateur graphique logique AND, on obtient des caractères de couleur 0. En utilisant l'opérateur graphique logique OR, on peut obtenir des caractères de couleur 15. Il faut alors modifier le dessin des caractères. Les matrices doivent être dessinées en couleur 15 sur un fond de couleur 0.

Affichage dans n'importe quel sens

La routine AFFTEXT présenté plus haut affiche les textes de la gauche vers la droite. Vous pouvez la modifier pour qu'elle affiche les textes vers le bas ou vers le haut en incrémentant ou décrémentant la position py d'affichage. La routine AFFTEXT2 affiche le texte en direction du bas. Après chaque affichage, la position courante d'affichage est augmentée de 8 pixels.

```
PROCEDURE afftext2(px,py,m$)
 LOCAL i
 LOCAL xaff
 LOCAL yaff
 LOCAL n
 '
 xaff=px
 yaff=py
 FOR i=1 TO LEN(m$)
  c$=MID$(m$,i,1)
  n=ASC(c$)
  IF LEN(car$(n))<>0
    PUT xaff,yaff,car$(n),1
  ENDIF
  ADD yaff,8
 NEXT i
RETURN
```

La routine AFFTEXT3 affiche le texte en direction du haut de l'écran. Après chaque affichage, la position y d'affichage est diminuée de 8 pixels.

```
PROCEDURE afftext3(px,py,m$)
 LOCAL i
 LOCAL xaff
```

```
 LOCAL yaff
 LOCAL n
 '
 xaff=px
 yaff=py
 FOR i=1 TO LEN(m$)
  c$=MID$(m$,i,1)
  n=ASC(c$)
  IF LEN(car$(n))<>0
    PUT xaff,yaff,car$(n),1
  ENDIF
  SUB yaff,8
 NEXT i
RETURN
```

Une variable SENS_AFFICHAGE peut contenir le type courant d'affichage. La routine générale d'affichage peut sélectionner une routine d'affichage spécifique en fonction du contenu de SENS_AFFICHAGE.

```
PROCEDURE affmessage(x,y,m$)
 SELECT sens_affichage
  CASE 1
    @afftext(x,y,m$)
  CASE 2
    @afftext2(x,y,m$)
  CASE 3
    @afftext3(x,y,m$)
 ENDSELECT
RETURN
```

En utilisant des méthodes plus complexes, on peut afficher des textes en suivant une forme géométrique, ou le bord d'une image. Le problème est de déterminer l'algorithme calculant la position d'une lettre quelconque.

Disparition progressive d'une image

En utilisant l'opérateur graphique logique AND, il est possible de faire disparaître progressivement une image. En affichant un masque fait de 2 couleurs (0 et 15) avec l'opérateur AND sur l'écran, on cache certaines parties de l'écran. Pour effacer progressivement une image, il faut afficher une série de masques avec de plus en plus de pixels de couleurs 0. Le dernier masque doit être entièrement en couleur 0.

```
PUT px,py,masque1$,1
PAUSE t%
PUT px,py,masque2$,1
PAUSE t%
PUT px,py,masque3$,1
PAUSE t%
PUT px,py,masque4$,1
```

Les instructions PAUSE servent à diminuer la vitesse d'effacement de l'image. Sans ces temporisations, l'effacement se ferait trop rapidement pour être visible. Les masques d'effacement peuvent être générés de manière aléatoire par une routine spécifique, mais aussi dessinés avec un logiciel graphique.

L'image est progressivement envahie par la couleur 0. Ce procédé d'effacement n'est donc valable que sur un fond de couleur 0. En remplaçant l'opérateur graphique logique AND par l'opérateur graphique logique OR, on obtient un effacement progressif en couleur 15. Cette autre technique n'est valable que sur un fond de couleur 15.

Apparition progressive d'une image

En inversant le processus, il est possible de faire apparaître progressivement une image. C'est une technique très utilisée par les programmeurs de jeu pour afficher les titres de leurs logiciels. En général, ces titres sont affichés sur un fond de couleur 0, donc avec l'opérateur graphique AND.
L'affichage est plus difficile à réaliser que l'effacement car il doit être réalisé dans un écran virtuel afin d'obtenir un bel effet visuel. Le programme suivant recopie l'écran dans un écran virtuel d'adresse adr_ecr%. L'affichage sur l'écran physique est réalisé avec l'instruction BMOVE. On peut aussi utiliser l'instruction RC_COPY.

```
t%=5
adr_ecr%=MALLOC(32000)
BMOVE XBIOS(2),adr_ecr%,32000
VOID XBIOS(5,L:adr_ecr%,L:XBIOS(2),-1)
'
PUT px,py,image$
PUT px,py,masque1$,1
BMOVE adr_ecr%,XBIOS(2),32000
PAUSE t%
'
PUT px,py,image$
PUT px,py,masque1$,1
BMOVE adr_ecr%,XBIOS(2),32000
PAUSE t%
'
PUT px,py,image$
PUT px,py,masque1$,1
BMOVE adr_ecr%,XBIOS(2),32000
PAUSE t%
'
PUT px,py,image$
PUT px,py,masque1$,1
BMOVE adr_ecr%,XBIOS(2),32000
VOID XBIOS(5,L:XBIOS(2),L:XBIOS(2),-1)
```

On peut aussi remplacer les masques aléatoires par des masques géométriques générés par programme. On peut ainsi afficher une image sous la forme de petits cercles ou de petits rectangles s'agrandissant pour reconstituer l'image complète. Cela peut aussi servir pour les effacements d'images. Avec des masques assez grands, on peut même faire une image qui s'affiche à partir d'un nom. Cela permet de faire un effet très spectaculaire, mais qui prend beaucoup de place mémoire.

Affichage de sprites de dimensions quelconques

Un sprite est une image qui peut s'afficher sur un fond quelconque. Contrairement à l'affichage en ombre chinoise, les sprites doivent

être de plusieurs couleurs. On ne peut afficher un sprite avec un seul opérateur logique. Il faut faire deux opérations graphiques logiques à la suite.
Un sprite est constitué d'un masque et d'une forme. La forme contient le dessin proprement dit du sprite, et le masque contient les informations sur les pixels de la forme devant être affichée. Le dessin du sprite est fait sur un fond de couleur 0. Le masque est un dessin de couleur 0 sur un fond de couleur 15. Les pixels en couleur 0 du masque correspondent à ceux de la forme devant être affichée.
La technique à mettre en œuvre est la suivante: l'image résultante doit être de la couleur de la forme si le masque est à 0, et de la couleur du fond si le masque est à 15. L'affichage d'un sprite se fait pixel par pixel. Le principe est toujours le même: si le pixel de masque est à 0, le pixel de forme est affiché; et si le pixel de masque est à 15, le pixel de l'écran n'est pas modifié. On arrive à ce résultat en deux étapes: le masque est d'abord affiché avec l'opérateur logique AND, puis la forme est affichée avec l'opérateur logique OR. L'affichage du masque se fait avec le mode logique AND. Les pixels de l'écran qui se trouvent en interaction avec les pixels du masque de couleur 0 se changent en couleur 0 (puisque n AND 0 = 0). Les pixels de l'écran qui entrent en interaction avec les pixels en couleur 15 du masque ne changent pas de couleurs (puisque n AND 15 = n). On a alors une image en ombre chinoise de la forme à afficher. L'affichage de la forme se fait ensuite avec le mode logique OR.
Les pixels de l'écran qui se trouvent en interaction avec les pixels de couleur 0 de l'image restent inchangés (puisque 0 OR n = n). Les pixels de l'ombre chinoise prennent automatiquement la couleur des pixels de la forme (puisque n OR 0 =n).

```
PUT px,py,masque$,1
PUT px,py,forme,7
```

Fabrication automatique des masques et des formes

Il n'est pas très facile de dessiner les formes et les masques des sprites. Pour éviter cela, le plus simple est de dessiner une seule image et d'écrire un petit programme générant automatiquement la forme et le masque du sprite. Cette technique est celle de la couleur transparente. Le sprite est dessiné avec un logiciel graphique. L'une des couleurs (généralement la couleur 15) est dite transparente, c'est-à-dire qu'elle ne sera pas affiché à l'écran et correspond aux pixels de l'écran qui ne doivent pas être modifiés par l'affichage du sprite.
Une fois le sprite dessiné avec une couleur transparente, il faut générer le masque et la forme. Le masque est généré en remplaçant la couleur transparente par 15, et les autres couleurs par 0. La forme est générée en remplaçant la couleur transparente par 0. Le programme suivant fabrique un masque et une forme à partir d'une image avec couleur transparente. Remarque: en raison du bug de couleurs du VDI, ce programme utilise la couleur 1 et non la couleur 15.

```
x1=10
y1=10
x2=40
y2=40
trans%=1
'
GET x1,y1,x2,y2,image$
@gen_masque
GET x1,y1,x2,y2,masque$
PUT x1,y1,image$
@gen_forme
GET x1,y1,x2,y2,forme$
END
'
PROCEDURE gen_masque
 LOCAL px,py
 '
 FOR py=y1 TO y2
  FOR px=x1 TO x2
   IF POINT(px,py)=trans%
     PLOT px,py,1
   ELSE
     PLOT px,py,0
   ENDIF
  NEXT px
 NEXT py
RETURN
'
PROCEDURE gen_forme
 LOCAL px,py
 '
 FOR py=y1 TO y2
  FOR px=x1 TO x2
   IF POINT(px,py)=trans%
     PLOT px,py,0
   ENDIF
  NEXT px
 NEXT py
RETURN
```

Une fois le masque et la forme stockés dans les variables masque$ et forme$, il ne reste plus qu'à les afficher ou à les sauver dans un fichier binaire pour une utilisation ultérieure.

Des sprites dans n'importe quelle résolution

On peut afficher des sprites dans n'importe quelle résolution à condition d'adapter le numéros des couleurs.
En moyenne résolution, les masques doivent être de couleur 0 sur un fond de couleur 3, et les formes sont dessinées sur un fond de couleur 0. En haute résolution, les masques doivent être dessinés en couleur 0 sur un fond de couleur 1 et les formes sur un fond de couleur 0.

En créant un sprite avec une couleur transparente, on perd cette couleur qui ne peut être utilisée dans le dessin du sprite.
Ce n'est pas très gênant en basse résolution, puisqu'il reste encore quinze couleurs. Par contre, il ne reste que trois couleurs en moyenne résolution, et 1 couleur en haute résolution. Il est donc préférable de dessiner les masques des sprites à la main (c'est-à-dire avec un logiciel graphique) pour la moyenne et la haute résolution. On évite ainsi de perdre une couleur.

DISQUETTE BEST OF GFA BASIC

Au lieu de perdre du temps à taper les listings des programmes, simplifiez-vous la vie en commandant la disquette BEST OF GFA, contenant tous les listings de ce numéro et des images Degas Elite utilisables avec les programmes graphiques.

Voir bon de commande en page 50

Le contenu de cette disquette est également disponible en téléchargement sur le serveur 3615 ATARI, sous la forme d'un fichier portant le nom de BESTGFA.TOS.

OFFRE SPÉCIALE

Des avantages supplémentaires exclusivement réservés aux abonnés de

Start micro magazine

Votre abonnement vous donne accès à "3615 MICROHELP(1)"

Votre abonnement vous procurera des réductions intéressantes sur le téléchargement et l'achat de disquettes de logiciels du domaine public

Votre abonnement vous fera bénéficier de tarifs promotionnels sur certains logiciels du commerce

Abonnez-vous et économisez immédiatement

80 F

Bulletin d'abonnement au verso

OFFRE SPÉCIALE

☐ OUI, je m'abonne à START MICRO MAGAZINE pour un an à partir du numéro à paraitre (11 numéros dont un double en juillet/août) au prix exceptionnel de **360 F**[2] au lieu de **440 F** (prix au numéro)

J'ai bien noté que je bénéficierai du service exclusif d'assistance télématique 3615 MICROHELP.

Ci-joint un chèque bancaire ou postal (exclusivement, pas de mandat) libellé à l'ordre de **FC PRESS**.

[1]les modalités de fonctionnement du service ainsi que votre code d'accès personnel vous seront communiqués dès l'ouverture du serveur.
[2]offre valable pour la France seulement. Etranger nous consulter.

Attention :
envoyer ce bulletin et votre chèque à :

FC PRESS
57, rue Danton
92300 LEVALLOIS-PERRET

Nom _____

Prénom _____

Adresse _____

_____

Code postal _____

Ville _____

Start Micro magazine
complétez votre collection

n°1

DOSSIER FALCON
Présentation du Falcon 030
Multimédia pour tous
TOS 4.0 et MULTITOS

SOS COURRIER
Les réponses à vos questions

ESSAI LOGICIEL
L'anti-virus Ultimate Virus Killer
Le traitement de texte Papyrus

GRAPHISME
Le logiciel de raytracing DKBTRACE
La couleur de A à Z

TABLEURS
Transférer vos données entre ST et PC.

MUSIQUE
Les messages MIDI

EDUCATIF
Vive l'école

COMPARATIF
Consoles de jeu portables

JEUX
Les jeux de rôle

INITIATION
Les différentes types de virus

PROGRAMMATION
3 nouvelles polices de caractères en Gfa Basic

TELEMATIQUE
Les nouveautés du téléchargement

LA DISQUETTE DU MOIS
49 accessoires indispensables

n°2

ACTUALITE
De nombreuses news Falcon

SOS COURRIER
Les réponses à vos questions

MUSIQUE
Les messages MIDI suite..

PORTFOLIO
Interface Midi Pocket Music

BUREAUTIQUE
Le tableur Graal Calc 3

EDUCATION
L'ordinateur est un jeu d'enfant

SYNTHESE D'IMAGES
Espaces de lumières
Logiciel graphique Cyber Shadows
Les nouveautés graphiques

GRAPHISME
Logiciel Persistence of Vision 1.0
Mode d'emploi de Néochrome Master

JEUX
Jeux d'aventures sur Micro

TEST MATERIEL
Fax Modem SUPREM-9624

DOMAINE PUBLIC
Sélection de 56 programmes à posséder absolument.

TELEMATIQUE
Les nouveautés en téléchargement
Les jeux PC en téléchargement

PROGRAMMATION
Ecrivez "un logiciel dont vous êtes le héros"
Animation dynamique en GFA Basic
Fichiers à accés directs en Omikron
Effets spéciaux graphiques en Assembleur

DISQUETTE DU MOIS
Le logiciel de dessin Néochrome Master 2.27
Une bibliothéque d'images

Pour commander, remplissez lisiblement ce bon (ou une copie) et envoyez-le accompagné du règlement à :
FC PRESS - 57, rue Danton - 92300 Levallois-Perret

Je commande le n°1 ☐, le n°2 ☐ à 38 F le numéro.
Soit ..... numéro(s) à 38 F = ... F + 15 F de frais de port et d'emballage (30 F pour l'étranger et les DOM-TOM) par chèque à l'ordre de FC PRESS.

Nom _____

Prénom _____

Adresse _____

_____

Code postal _____

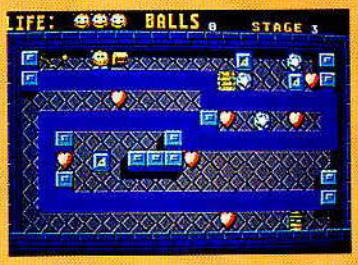
Ville _____

Pays _____

Start Micro Magazine

n°3
JANVIER 1993

LA MODELISATION DES OBJETS 3D



LE MULTIMEDIA A LA MAISON

JOURNAL L'ALTERNATIVE FANZINE

INITIATION SAUVEGARDEZ VOS FICHIERS

TEST LOGICIEL PARALLELE 3D

SOS LES REPONSES A VOS QUESTIONS

DISQUETTE INCLUSE

* 10 JEUX COULEURS
* 4 JEUX MONOCHROMES
* 8 UTILITAIRES PRECIEUX
* LES LISTINGS DU JOURNAL

LES IMAGES DES DEMONSTRATIONS FALCON

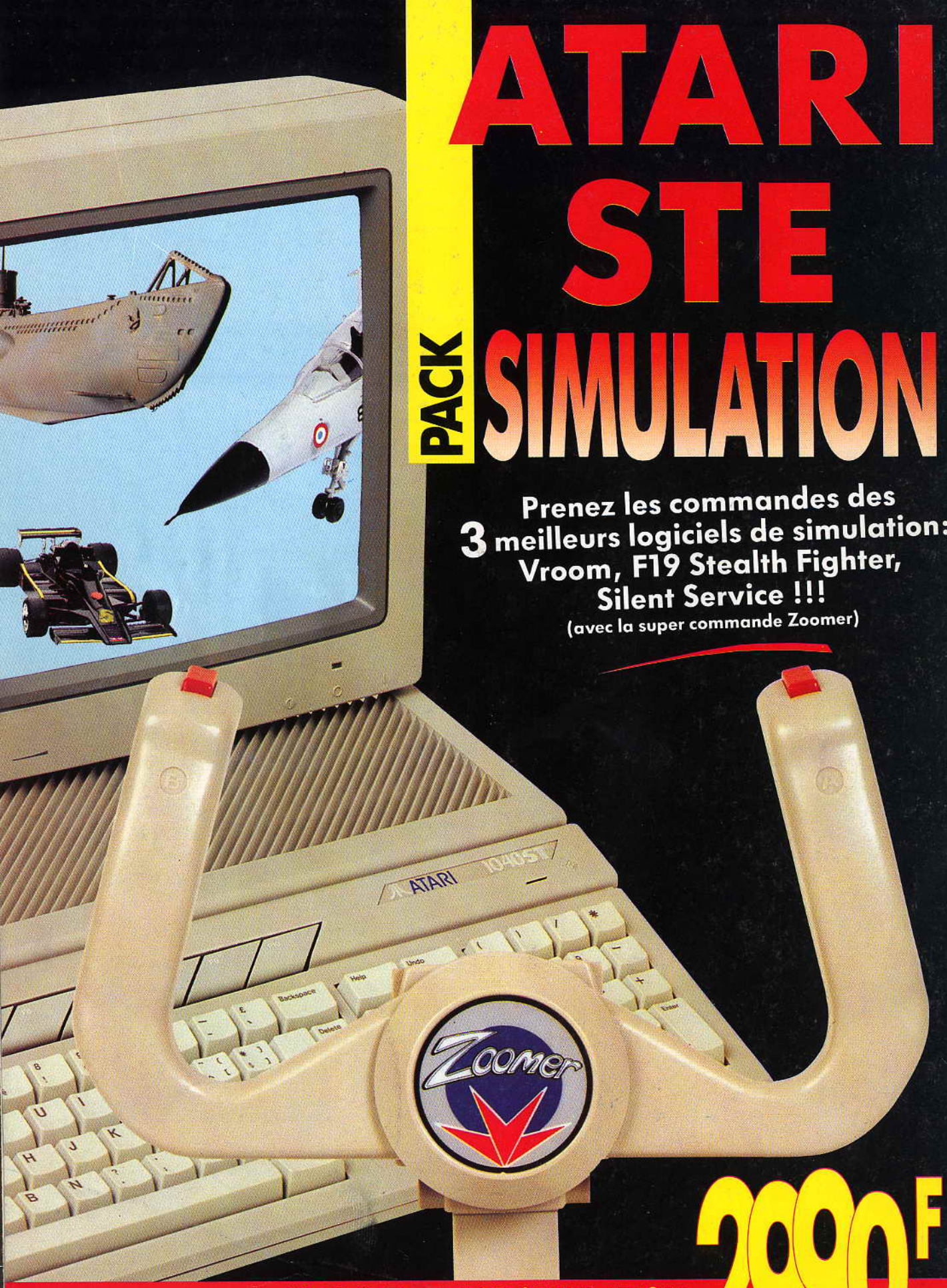
ACTUELLEMENT EN VENTE EN KIOSQUE

ATARI STE

PACK SIMULATION

Prenez les commandes des 3 meilleurs logiciels de simulation: Vroom, F19 Stealth Fighter, Silent Service !!!

(avec la super commande Zoomer)



ATARI 1040 STE + PACK SIMULATION 2990F TTC